DATA MANAGEMENT FOR KNOWLEDGE-BASED SYSTEMS

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston -- University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Kam Man Law

August, 1987

Acknowledgements

I wish to express my sincere thanks to my thesis advisor Dr. Christoph Eick for his patience and valuable guidance. My thanks also go to the committee members Dr. Scamell and Dr. Elmasri. Special thanks are extended to Dr. Davidson and Dr. Scamell for reviewing and editing my thesis.

I am also indebted to Bonnie, Iris, Dorothy, and Lex who took messages for me throughout the last year of my graduate study.

The biggest thanks of all go to my grandmother; my parents; my brothers, Kam Wah, Kam Wai, and Paul; my sister, Lai Chun; and Paul's wife, Stella. Without their encouragement, financial support, patience, and love, completion of this goal would have been impossible.

DATA MANAGEMENT FOR KNOWLEDGE-BASED SYSTEMS

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston -- University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Kam Man Law

August, 1987

Abstract

In recent years, expert systems have gained a large popularity in Computer Science as a result of the improvement made in Artificial Intelligence research and the announcement of the fifth generation computers. However, the design and implementation of large computerized knowledge bases have raised new data management problems.

This thesis explores the new problems facing conventional Data Base Management Systems (DBMSs). It surveys the knowledge representation in Databases, Artificial Intelligence, and Programming Languages in order to search representation schemes for DALI, a Knowledge Base Management System. The concepts of DALI are discussed and the data model S-diagram used for knowledge base design is described. Furthermore, the features of DALI are compared with those of DBMSs; the advantages and disadvantages of DALI are examined. In its first version, the framework of DALI contains a schema compiler, a pattern matcher, and a storage structure program. The design and implementation of these essential components are described in detail.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of the expert systems and the fifth generation computers, database management systems have faced new problems. These problems include (see also [Eick86]):

1. Knowledge Bases (KBs) in expert systems contain not only facts, but also rules and control knowledge (i.e., objects that have inferential capabilities). However, conventional database management systems can only handle facts concerning the Universe of Discourse (UoD).
2. Most expert systems are written in LISP; therefore, data management systems must be capable of handling S-expressions, the data structure for LISP. But so far, no such system is available for handling large computerized LISP-based applications.
3. Knowledge bases usually contain uncertain knowledge. But conventional data management systems do not provide special features to handle these "fuzzy" data.
4. Algorithms like forward chaining, backward chaining, and pattern matching are frequently used for accessing data in the KB. Nevertheless, conventional database management systems do not provide special data structures suitable for an efficient implementation of these

algorithms.

5. The internal representation of data in a knowledge-based system might change with time for performance reasons. When the organization of the physical layer changes, it should not affect the user interface layer. This can be accomplished by implementing abstract data types in different levels. Nonetheless, many database systems do not support this capability.
6. Rules stored in knowledge bases are usually very complex -- tens to hundreds of conditions are quite usual, and manual checking of rule consistency would be very time consuming. However, conventional database management systems do not contain any automated tools for enforcing rule integrity in the knowledge bases.

One way to solve these problems is to enhance the capabilities of conventional database management systems so that they handle the "fuzzy" data as well as the knowledge base rules. An alternative to this is to develop a new data management system with designs pertinent to the needs of the expert systems. We will call this new system as Knowledge Base Management System (KBMS). In the last few years, some ad hoc solutions have been found to solve the above mentioned problems. However, a generalized system which can handle any large class of knowledge base applications has

yet to be created.

This thesis discusses the concepts of a knowledge base management system called DALI (an acronym for Data-management for LISP-knowledge-bases). DALI is a research project with an objective to study the data management problems in knowledge bases.

This thesis also reports the development of the first KBMS prototype. The system developed in this thesis consists of: a schema compiler, a pattern matcher, and a storage structure program. The schema compiler checks the syntax and semantics of a schema and generates, if the schema is error free, a program which contains all the Knowledge Base Manipulation Functions (KBMFs) that are declared as predefined operations in the schema (a schema is where the data classes and simple value sets are defined). The pattern matcher is used to match patterns specified in the Knowledge Base Manipulation Language (KBML) against the data in the KB during data retrieval. It is also used to match patterns specified in the schema against the values given in the KBML during type checking. Finally, the storage structure program serves as an interface between the KBML and the KB. These essential components constitute the central issue of the thesis. A synopsis of the chapters is

outlined in the following paragraphs.

In chapter 2, we survey the knowledge representation of databases, artificial intelligence, and programming languages. The purposes are two-fold: to review the current research and to select representation schemes for the DALI system.

Chapter 3 describes the basic concepts of DALI. It also explains the functions of its components and discusses the advantages and disadvantages of using this knowledge base management system. Lastly the features of DALI are compared with those of DBMs.

Chapter 4 introduces the data model of the KBMS. It also examines the syntax of the KBML.

Chapter 5 covers the features and the implementation of the pattern matcher. The pattern matching language is applied in both the schema language and the knowledge base manipulation language.

Chapter 6 discusses the design and implementation of the schema compiler as well as the data dictionary. This chapter also shows how to use the schema language to define

a conceptual schema. The term "schema language" is used rather than "data definition language" because the former is more meaningful pertaining to the conceptual schema.

Chapter 7 describes the design and implementation of the last component of DALI -- the storage structure program. In addition, the selection of data structure for the DALI-KB is discussed and the features of the Xerox 1186 Artificial Intelligence System, in which DALI is implemented, are mentioned.

Chapter 8 gives the summary and conclusions of the DALI system. It also outlines future research on DALI.

Since terms used in the database literature have not been standardized, we will follow the terminology and standardization suggested by the ISO workgroup WG3 [Grie82] in this thesis.

Chapter 2

A Review of Knowledge Representation

With the ever-increasing complexity of computer systems, researchers are searching for development tools, techniques, and high level concepts for representing knowledge in every area of Computer Science, particularly in Artificial Intelligence (AI), Databases, and Programming Languages.

In the following sections, we will give an overview of Knowledge Representation in these three areas. First, we will study the tools and techniques that are used to describe objects, operations, and constraints in Databases. Next, we will turn to Knowledge Representation in Artificial Intelligence. As Knowledge Representation is the central issue in AI research, we will discuss the Representation Schemes [Haye74]. Last but not least, we will look at some programming techniques that are dominant in Computer Science.

2.1 Databases

For many years, data models have been used to describe

entities that are perceived in the Universe of Discourse as well as relationships between them, before going into a detailed logical and physical database design. Data models are a collection of mathematically well defined concepts; they are used to help define attributes of, operations on, and relationships among objects of the real world that have to be expressed in a computer. In addition to these, they also help describe integrity rules over the objects and their operations. For instance, an integrity rule on an employee database may state that the date of resignation must be later than the date of employment. Data models also provide a formal basis for tools such as Data Definition Language (DDL), Data Manipulation Language (DML), and Query Language (QL) that are implemented by a database management system (DBMS).

Earlier data models stress the form of data that would facilitate for storage and/or manipulation in a computer; that is, good performance gained a large attention. Furthermore, the access path structure was also emphasized so that retrieval of information could be optimized. This group of data models constitute the classical data models. The classical data models are the hierarchic, the network, and the relational data models.

Hierarchic data models represent objects in a tree structure, using one to many binary relationships. The advisors and students relationship provides an example of a hierarchy. An advisor may advise many students, but a student can only have one advisor.

Network data models are the superset of hierarchic models -- children in a tree can have multiple parents. An example of this is class registration. Students may register for many classes, and each class may contain many students. Therefore, both the relationship student to class and the relationship class to student are one to many relationships.

Relational data models [Codd70] are based on the mathematical concept of a relation. A relation is a two-dimensional table where each column is referred to as an attribute and each row is referred to as an n-tuple. A relation can be used to describe both objects and many to many, n-ary relationships. No two rows in the relation are identical and the order of the rows is not important.

In recent years, data models have stressed the meanings (semantics) of the information; hence, semantic rules for objects and their operations play significant roles in these

data models. These models are called semantic data models. Semantic data models comparatively provide more modelling powers than the classical data models do; that is, they can capture more meaning with their richer and more expressive concepts.

2.2 Artificial Intelligence

The basic problem of knowledge representation in AI is the development of a representation scheme [Haye74] with which to specify a knowledge base. The early designs for knowledge representation emphasized heuristic search techniques; whereas, the current designs stress for the need for storing expert knowledge together with control knowledge in a system. The well known representation schemes [Brod84] are logical, network, procedural, and frame-based. Each is discussed in the following sections.

2.2.1 Procedural Representation Schemes

Procedural Representation Schemes represent knowledge in terms of a collection of active agents or processes. They are influenced mostly by LISP. In fact, LISP once was a favorite representation scheme due to its symbolic nature. Procedural schemes beyond LISP involve control structures

and activation mechanisms offered for processes. The advantage of these schemes is that the need for wasteful search is eliminated because they allow the specification of direct interactions between facts. The major drawback is that procedural knowledge bases, like programs, are hard to understand and modify. This category is represented by PLANNER.

2.2.2 Logical Representation Schemes

Logical Representation Schemes are almost counterparts of Codd's Relational Model [Codd70] in Database Management. They represent knowledge by means of logical formulas which are composed of constants, variables, functions, predicates, logical connectives, and quantifiers. Their advantages are

* Availability of inference rules
* Availability of clean formal semantics
* Simple notation
* Economic representation of knowledge

On the other extreme, their drawbacks are

* Lack of organizational principles needed in the knowledge base
* Difficulty in representing procedural and heuristic knowledge

Several languages have combined logical schemes with others to gain more advantages. For example, both PROLOG and FOL combined logical schemes with procedural schemes.

2.2.3 Network Representation Schemes

Network schemes exist in a wide variety of forms. In general, they represent knowledge in terms of a collection of objects (nodes) and binary relationships (edges) between them.

Network Representation schemes offer organizational principles to a knowledge base. These principles include: Classification, Aggregation, Generalization, and Partitions. They also offer a good scheme for information retrieval due to their path nature. Further, their graphical notation makes it easier to implement a network knowledge base.

The only major drawback of network schemes is the lack of formal semantics and standard terminology.

Both PSN and KL-ONE are built in part by the network schemes.

2.2.4 Frame-Based Representation Schemes

Frame-Based Representation Schemes were first introduced by Minsky [Mins75] in 1975. He proposed to combine ideas from semantic networks, procedural schemes, linquistics, etc. to develop a new representation scheme. The Frame-Based Representation schemes are a collection of complex data structures called frames. Each of these frames has slots for objects. Different kinds of information are stored in these frames, e.g. the default values for their objects and the actions for any unexpectancy. The structural form of these frames follows some of the organizational principles and the "looser" principles as well. An example of a "looser" principle is the notion of similarity between two frames.

FRL KRL and KL-ONE all contain frames in their knowledge bases.

2.3 Programming Languages

Knowledge Representation in programming languages can be classified into four paradigms: procedure-oriented, object-oriented, data-oriented, and rule-oriented.

2.3.1 Procedure-Oriented Programming

In procedure-oriented programming, procedures, which are composed of instructions, when invoked, will change the shared data structure (e.g. a knowledge base) independently. Procedures are considered as active entities because they can create side effects in the shared data structure; on the other hand, data are treated as passive entities because they are being manipulated by procedures. Most of the programming languages today are procedure-oriented. For instance, LISP and its dialects, such as INTERLISP, FRANZLISP, ZETALISP, etc., are all paradigms of procedure-oriented programming languages.

2.3.2 Object-Oriented Programming

In object-oriented programming, actions are invoked by sending messages between objects belonging to one or more types. A type is a template which holds data and operations for its instances. It can be created not only by specifying data objects and methods in it, but also by inheriting from other types (usually called super types). Once a type is defined, the instances (or objects) of it can be created from then on.

Actions are defined as methods in a type and operations in object-oriented programming are performed by sending messages. When a message is received by an object, the message will be checked against the selectors and the corresponding method will be invoked. The implementation of methods are usually isolated from where the selectors are. The isolation of the actual implementation of methods has several advantages:

(1) It offers top-down design methodology with successive refinement.

(2) It allows program verification without the actual working environment.

(3) It eases program maintenance.

Smalltalk, derived from SIMULA, is the pioneer of object-oriented programming. Its inheritance is in hierarchical form -- the simplest case of network, each class specifies only one super class. Nonetheless, some may support network inheritance. Symbolics 3600 [Symb84] and LOOPS [BoSt83] fall in this paradigm.

The Symbolics 3600 system uses ZETALISP, a dialect of LISP, as its principal language. Object-oriented programming techniques, which deal with instances of types and generic operations defined on those types, are used

throughout the system. Types in the 3600 system are abstract types known as flavors; whereas, the objects, which are instances of types, are called flavor instances (see Fig. 2.1)

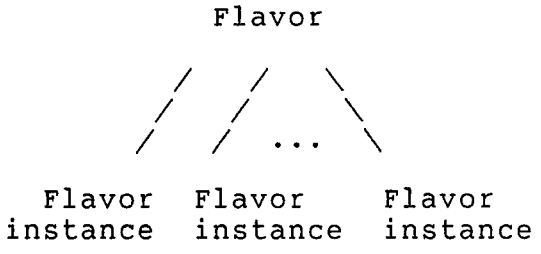

Fig. 2.1 Hierarchical Structure of Symbolic 3600 Objects

Flavors are classified as "base" flavors, "mixin" flavors, and "user-defined" flavors. Base flavors serve as the foundation for building a flavor family while mixin flavors serve to implement particular needs of other flavors. Finally, user-defined flavors are built out of the base and mixin flavors to suit the user's applications.

Symbolics 3600 provides support for object-oriented programming through a collection of language features known

as the Flavor system. With the Flavor system, the users define the flavors -- one of the many user-defined types in ZETALISP -- and methods, which are generic operators, associated with them in one part of a program. Then, in another part of the program, instances of the flavors are instantiated and manipulations of the instances are performed by sending messages which request that specific operations be performed.

A flavor definition contains information about instance variables, methods, names of its component flavors, and declarations of relationships and interdependencies with other methods and flavors. Methods and instance variables can be local to objects or inherited from component flavors. When a new flavor definition is built from component flavors, the method definition for this flavor has control over the methods from the component flavors.

LOOPS which adds object-oriented programming and more to the procedure-oriented programming of INTERLISP is worth mentioning in this programming technique. It provides classes and instances as INTERLISP file objects. It also provides user extendible property lists which store documentation and information in classes, variables, and methods. Moreover, it provides composite objects which work

as follows. Given templates for related objects, they are instantiated as a group.

Unlike Symbolics 3600, LOOPS has three kinds of objects; namely, instance objects, class objects, and metaclass objects. They form a hierarchical structure and are depicted in Fig. 2.2.

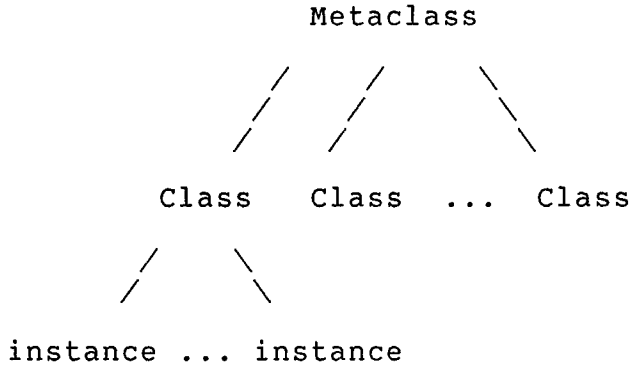

Fig. 2.2 Hierarchical Structure of LOOPS Objects

From Fig. 2.2, it can be easily seen that the instances of a class are instances and the instances of a metaclass are classes. A class contains information about instance variables, class variables, methods, and a list of other

classes called "super classes". While instance variables are used to specify default values to its instances, class variables may be used by methods.

Methods, instance variables, and class variables can be inherited recursively from a super class or all its multiple super classes. In the case of inheritance of multiple super classes, names conflict is resolved by using a depth-first left to right precedence. For example, if A has a superlist (B C), and B has a superlist (X Y), the inheritance order will be B, X, Y, and C. Suppose X has a method M1 for selector S1, and C has M2 for S1, M1 from X will be used instead of M2 from C. In fact, any super of B with a method for S1 has precedence over the one from C. Although inheritance for methods and class variables are made at run time, it is not necessary true for the instance variables.

Private instance variables which are not shared with other instances of the same class must be defined directly by the users.

Operations are invoked by sending messages. Messages received are checked against the selectors. When matches are found, the corresponding methods will be invoked which might cause variables changed as a side effect.

Another programming language worth mentioning is Ada. Ada is an all-purpose language; its capabilities outperform any other programming languages in common use in many areas. Ada provides object-oriented programming through the features of generic program units and packages [Youn83].

A package allows the specification of a set of logically related entities. It consists of two parts: a package specification and a package body. The specification part declares the entities which are visible outside of the package. In general, it includes constants, variables, types, and the specification of program units, such as subprograms, tasks, and packages. On the other hand, the package body contains the implementation of the entities declared in the specification part. However, entities in this part are not visible outside the package body. Since a package encapsulates a set of data objects and their related operations, it is easy for it to implement abstract data types and various system resources such as common data pools, I/O buffers, etc.

A generic program unit is a template for creating instances of that unit. The instance creation is called generic instantiation and it happens during the compile-time. A generic unit allows parameter passing so

that the characteristics of the instance can be determined. For example, in its simplest case, the instantiation of a stack needs the supply of a name as the actual parameter to be identified by the program.

The way Ada works as object-oriented programming is that templates are defined as generic package units -- all the data objects and the operations are defined inside these units. A copy of package is made via the generic instantiation in the compile-time. During generic instantiation, a name is given to the new copy and the actual parameters will substitute for the formal parameters. When compilation is done, objects should have been created. Then operations on these objects are invoked by procedure call statements in the run-time.

2.3.3 Data-Oriented Programming

In data-oriented programming, a mechanism is needed to invoke a procedure or function when a special variable is changed.

Data-oriented programming is suitable for one program to monitor the behavior of another program. Since both programs are isolated from each other, modification of codes

in one program would not affect the other.

Indeed data-oriented programming has an opposite effect to object-oriented programming. In data-oriented programming, procedures or functions are invoked as a side effect when data are changed while in object-oriented programming, variables are altered as a side effect when messages invoke procedures to perform operations.

Loops includes both of these mechanisms and more in its programming environment.

LOOPS uses the mechanism active value, in data-oriented programming, to trigger the invocation of a predefined procedure or function. Every time when access to a variable occurs, LOOPS will check whether the variable has an active value. If so, the specified procedure or function will be invoked for actions.

As mentioned above, data-oriented programming is appropriate for interfacing between independent processes. It may be well-suited for writing simulation programs. For example, in a traffic simulation, one can create a viewer, an independent process from the traffic simulation, which will update the display positions of automobiles. Suppose,

active value, say POSITION, is being defined as its traffic coordinate system for each automobile. This active value will invoke the procedure to send update messages to the viewer when simulation process puts a new value into POSITION. As another example, one can apply active values to a debugging program in order to keep track of references to particular variables.

2.3.4 Rule-Oriented Programming

Rules are simply condition-action pairs. They specify actions to be taken when certain conditions are satisfied. Unlike other programming paradigms, rules make it convenient for describing flexible responses to a wide range of events. Rules, in this programming environment, are usually arranged in a predefined order set called a production system. A production system contains control structures which affect the behavior of actions; that is, they make the decision of which productions to fire.

Generally, all production system languages share the following common characteristics:

(1) Every production system language uses conditional statements called productions.

(2) The interpreter which is made up of high level

functions has access to data memory, production memory, and state memory. Production memory stores all the productions (i.e., conditional statements) and any static relations (e.g. a linear ordering) between productions; it should have no size limit. Data memory is a temporary workspace that stores the currently relevant knowledge processed by the productions and the static relations between the data. Unlike production memory, the size of data memory is limited. State memory stores the information that both data memory and production memory are not responsible for. For instance, information like the name of the last production to fire can be stored in state memory for reference purposes.

(3) The interpreter in a production system language repeatedly matches the productions in production memory against the data in data memory and the information in state memory, then it chooses the productions with TRUE antecedents to fire; as a result, changes in data memory will occur. The interpreter itself is also responsible for effecting changes to state memory when necessary. This is called the recognize-act cycle.

Production system languages use different strategies to

fire productions. Some production system languages may execute the actions for every satisfied condition on each cycle, some may use simple decision procedures to select some or all productions to fire, and some may even use complex decision procedures with more intelligence to choose which production to fire.

Some languages allow productions, in the selection process, to read state memory as well as data memory so that they have the role in choosing which productions to fire. On the contrary, others allow only the interpreter to have access to state memory.

Production system languages have been used for building expert systems for many years. Some of the well-known production system languages are OPS [Schm77] and AGE [Aiel81].

OPS is a production system language designed for the Instructable Production System (IPS) project at CMU, which attempted to answer the question of how suitable the production system representation is for large general problem solving programs. It has all the common characteristics of production system languages mentioned above. But, the production system does not have access to

state memory. This restriction causes some problems to the language. The obvious one is OPS becomes slow because conflict resolution in the selection process is solely handled by the interpreter.

The OPS interpreter performs an exhaustive search of productions to insure that an instantiation of a production will be found when there exists one. Among all the legal instantiations of all productions are found on each cycle, only one is selected for execution. The interpreter is responsible for handling conflict resolution since the information in state memory is not open to the productions. In performing conflict resolution, OPS makes use of five rules. The first rule is always executed, and the rest of the rules will be applied, on the condition that the first rule is successful, in the given order until all but one have been rejected.

In summary, the first rule guarantees a fair chance for each instantiation of production. The second rule chooses the instantiation with the most recency. The third rule is the extension to the second rule by which the instantiations with the greatest number of condition elements are more preferable. The fourth rule gives preference to the instantiation of the most recently created production.

Finally the last rule randomly selects an instantiation.

Actions in OPS are simple functions that will modify the contents of data and production memory. OPS includes a complete set of primitive actions, assert and delete, that would effect the changes to the contents of data memory. The way that OPS manipulates lists is comparable to that LISP does.

Three primitive functions, READP, BUILD, and EXCISE, are given to the system in order to perform self modification. READP brings the production into data memory, EXCISE deletes it from production memory before OPS modifies the production in data memory with its general processing capabilities, and finally BUILD deposits the modified version to production memory.

For input and output, OPS has two functions, READ and WRITE, to interact with the outside world. READ accepts data from the users and places them in data memory. In contrast, WRITE puts the instantiated forms to the users' terminal.

AGE is a software tool which contains knowledge about constructing knowledge-based programs. Therefore, AGE

itself is a knowledge-based system. It came out of Standford University and is implemented in INTERLISP. It was initially designed for AI scientists who were familiar with current problem solving techniques and production-rule representations of knowledge. Its framework is a collection of building block programs with an intelligent front-end to guide the users in building knowledge-based programs. This process involves augmentation and modification of the framework provided by AGE.

AGE employs the Blackboard model which allows incremental hypothesis formation. The Blackboard model has been used in HEARSAY-II [LeEr77] and CRYSALIS [EnNi77]. A blackboard-based program consists of three components, the Blackboard, the Knowledge Sources (KS's), and the Control.

A blackboard is a global data base; it is used as a medium for communication and interaction among the KS's. The hypotheses in the blackboard, which are generated by inference rules in the KS's, are organized in a hierarchical structure. In general, the blackboard represents the analysis level of task domain hierarchically.

Knowledge Sources which are represented as sets of production rules contain the knowledge, provided by the

users, of the task domain. They respond to the changes in the blackboard.

Rules in the KS's consist of a left-hand-side (LHS) and a right-hand-side (RHS). The RHS will be fired when the set of conditions in the LHS is evaluated to TRUE. Fired rules will take one of the follow actions.

(1) PROPOSE a change in a hypothesis

(2) EXPECT some changes may occur in the hypotheses

(3) ACHIEVE a particular value or a state

Each inferential rule generated by the RHS is assigned a certainty value or probability by AGE. Moreover, "single" or "multiple" hit strategy can be used for the rules.

The control components contain mechanisms that allow the users to (a) invoke the KS's conditionally and (b) select interested items off the blackboard. They are often considered to be a higher level KS which can manipulate a set of domain-specific KS's. Above all, they are responsible for invoking appropriate problem solving methods and of course appropriate KS's in a specific situation as well.

AGE has been used to rewrite some programs: CRYSALIS,

PUFF (with two different versions), and EMYCIN. The results are very satisfactory. In its long range goal, AGE will help people with less AI knowledge write knowledge-based programs.

Chapter 3

DALI -- A Knowledge Base Management System

DALI [Eick86] is a research project on data management of large computerized knowledge bases (KBs). It attempts to solve the problems facing knowledge base management systems.

In this chapter, we will first introduce the prototype of DALI and explain the functions of the essential components of it. Then we will discuss the advantages and disadvantages of using DALI. Lastly, we will examine the similarities and the differences between DALI and a DBMS.

3.1 Introduction to DALI

DALI, a LISP-based data management system, uses a semantic-net-like knowledge representation approach. It is intended to solve new data management problems that arise from large computerized knowledge bases which describe the expertise of specific applications such as cardiac decease, computer system configurating, etc.

A DALI-KB consists of a set of classes. Classes are defined by way of the schema language; they can be arbitrary

data types. A class may be a subset of other class, and all classes are subsets of Class KB. Furthermore, a class may overlap with other classes.

Classes contain entities. Entities are characterized by their attributes. Attribute values distinguish class members. In DALI, entity numbers are used to refer to entities. An entity number functions as a unique identification to an entity; it will not be changed throughout the life of the KB.

Attributes assign members of a range class to members of a domain class. Attributes are divided into two groups: simple attributes and role attributes. Simple attributes are lexical types (see chapter 4) and role attributes are non-lexical types.

DALI provides a schema language, a pattern matching language, a knowledge base manipulation language, and a storage structure language. The schema language is used to define simple value sets and data classes. It can also be used to specify restrictions on memberships of class. For example, if a class is limited to one hundred members, DALI will automatically check this restriction whenever a new member is created in that particular class.

The pattern matching language is used to define patterns in simple value sets and attribute constraints. It is also used to specify selection criteria in the knowledge base manipulation language.

In DALI, the knowledge base manipulation language and the storage structure language form a two-layer architecture between the end users and the KB. The KBML is a high level interface language used by the users for accessing the data of the KB while the storage structure language receives calls from the KBML to change data in the KB physically. The advantage of having a multi-layer architecture is that the internal representation of DALI is transparent to the user; therefore, principle of information hiding is achieved.

The knowledge base manipulation language is composed of a set of high level data manipulation functions -- we will call them KBML-functions. These functions can be embedded in a LISP program or applied interactively on the top level of the data management system.

With the KBML-functions, KBs can be loaded into or unloaded from the memory. The advantage of having the KB residing in the memory is that data accessing would be much

quicker. New entities can be created and assigned entity numbers to them. They can also be connected to an already existing entity since entities may be members of more than one class. When entities are no longer valid in a class, they can be eliminated. Attribute values in an entity can be replaced by new values. Furthermore, attribute values can be appended to or deleted from an entity. The KBML provided by DALI can also be used for data retrieval and KB navigation. In a DALI-KB, the values of role attributes are entity numbers; therefore, by using the entity numbers, one can navigate from one class to another. The data manipulation language also allow sequential read and print of a class.

DALI also observe data integrity in the knowledge base. Whenever an entity is created in or deleted from the KB by the KBML, consistency rules are enforced automatically. This is very important because if contradictions or inconsistencies occur in the KB, the inference engine may deduce contradictory answers for different queries.

3.2 Essential Components of DALI

In its first version, the framework of DALI is composed of a schema compiler, a set of storage structure functions,

and a pattern matcher. These components and their inter-relationships are shown in Fig. 3.1. The schema compiler checks the syntax of a schema. If no error occurs in the schema, it will produce an entry containing the intermediate codes of the compiled schema and store it in the data dictionary, a global variable. These intermediate codes will be used by the KBMF-code generator (code generator for short) which is a sub-component of the schema compiler. The code generator generates a program which contains all the allowed data manipulation functions that will be used by the end users for accessing the KB. The data manipulation functions produced contain LISP expressions that will validate the arguments and check the consistency of data. When validations are done and no infringement occurs, the KBML-functions will call their corresponding storage structure function to perform the job physically. Some data manipulation functions may contain patterns as arguments. In that case, the pattern matcher is called upon to perform pattern matching.

All these components mentioned above are essential to DALI; they are also the focus of this thesis and will be discussed in detail in the later chapters. In this section, we will briefly describe the functions of each component.

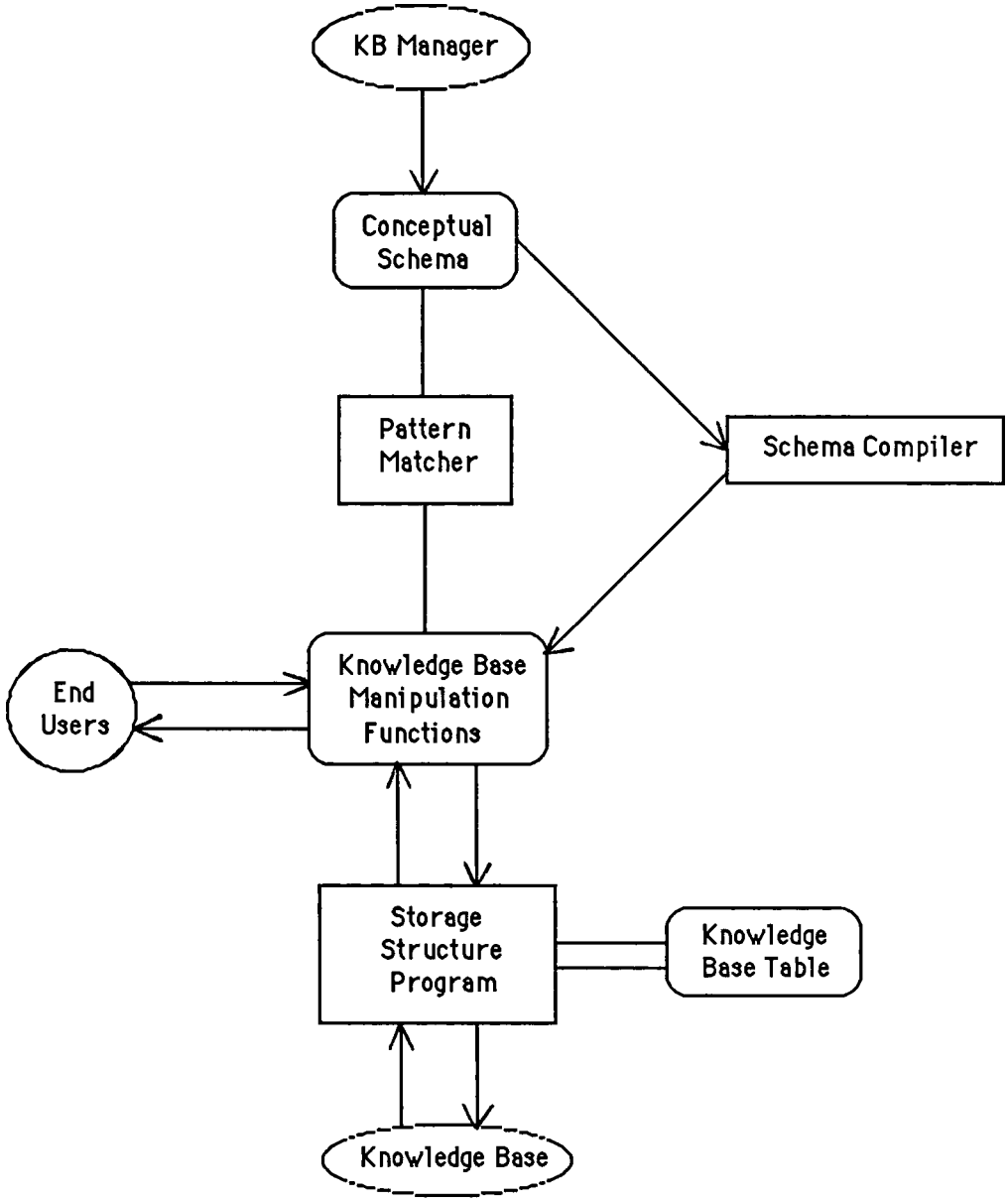

Fig. 3.1 Inter-relationships of DALI components

3.2.1 Schema Compiler

The schema compiler accepts as input the conceptual schema and produces as output a program that contains a set of KBML-functions. Its functions include

1. checking the syntax and semantics of a conceptual schema (CS)
2. building a data dictionary entry based on the CS
3. creating a listing for the schema
4. printing diagnostic messages on a listing
5. generating high level functions which serve as an interface between the user and the storage structure programs.

3.2.2 Storage Structure Program

The storage structure program is a set of low level data manipulation functions; they implement the high-level KBML in the respect of data operations. Its functions are:

1. to manipulate the physical data of the KB
2. to hide the internal representation of the KB from the user
3. to protect the KB from any intentional or unintentional destruction of the physical data
4. to serve as an interface between the KBML and the

physical data.

3.2.3 Pattern Matcher

The pattern matcher does pattern matching in both the schema language and the KBML. The functions of it are:

1. to do data matching during data retrieval from the KB
2. to do type checking during entity instantiations.

3.3 Advantages and Disadvantages of DALI

Like any other system, DALI has its advantages and disadvantages. But its advantages overshadow its limitation. The advantages of DALI are given below:

1. Object-oriented programming is supported; therefore, arbitrary data types can be defined by the conceptual schema language.
2. Data integrity is done by the KBMS automatically; as a result, user programs become shorter.
3. Information hiding is attained because the user does not know the internal representation of the KB.
4. Internal representation can be changed without any changes in the user program.
5. The maintenance cost is decreased because the centralization of the data management functions results

in less redundancy in codes as well as data.

On the other extreme, its disadvantages are summarized as follows:

1. The user may lose some freedom due to the restrictions enforced by the KBMS.
2. Smaller applications may not gain the advantages because of the software overhead.
3. The structure of a project may be altered due to the fact that data management functions are centralized.

3.4 DALI vs. DBMS

DALI shares four basic similarities with a DBMS.

1. A schema language is used to define the semantics of the conceptual schema.
2. A set of data manipulation functions is used to navigate or manipulate the objects of the database.
3. Some mechanisms are implemented into the data management system to insure data integrity in the database.
4. The conceptual view of data is separated from the physical view.

On the other end, DALI is different from a DBMS in

three major ways:

1. DALI is implemented by LISP since AI-programs are mostly written in LISP.
2. DALI supports semantic-net-like representation, widely used in AI-programs.
3. DALI uses a different approach in the use of data dictionary. The data dictionary in a conventional DBMS is used during the run-time by the DBML while our approach is that the data dictionary is used during the compile-time to help generate a KBML program and is not used again during the run-time of the KBML.

Chapter 4

The Data Model for DALI

Data models have been used to define the semantics of entities that are perceived in the Universe of Discourse. They are a collection of mathematically well-defined concepts that are used to help define attributes of, operations on, and relationships among objects as well as integrity rules over the objects and their operations.

The data model we will be using for knowledge base design is called S-diagram [EiRa85]. It is a graphical data model which was influenced by the binary relation model [Abri74] and the SDM [McLe78].

4.1 Concepts of S-diagram

The example given in Fig. 4.1 provides a high level representation of the concepts of S-diagram.

A class in S-diagram is represented using an oval, and it can be either of the two kinds -- lexical or non-lexical. A lexical class is a primitive data type such as INTEGER or REAL. However, a knowledge base is usually LISP-based, thus

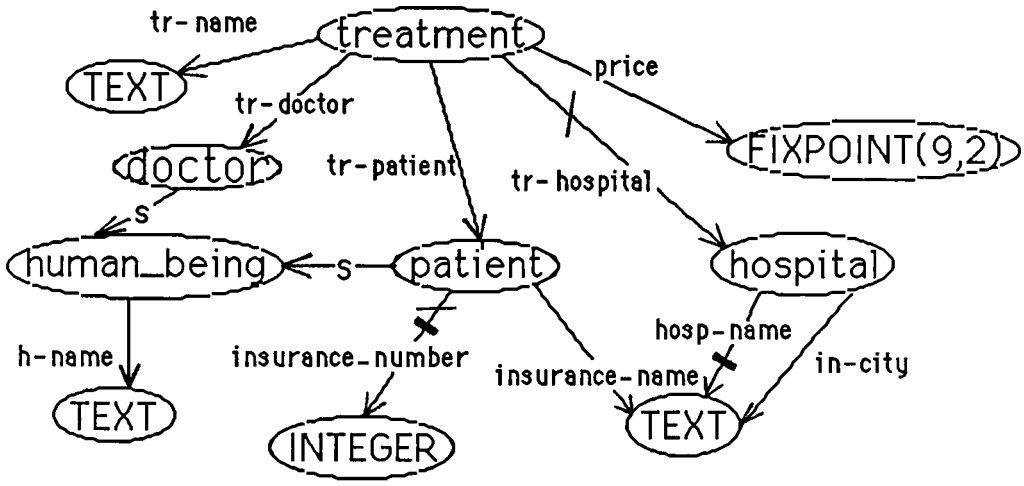

Fig. 4.1 S-diagram of Treatment

an additional data type called SEXPR is taken into account. SEXPR can be either type ATOM or type LIST. A lexical class in S-diagram is denoted with its name in capital letters.

On the contrary, a non-lexical class is not any of these primitive data types, e.g., class hospital in Fig. 4.1. So, in order to define this type and store it in a computer, we need to describe its attributes (characteristics of a class) in terms of lexical classes. A non-lexical class in S-diagram can be easily recognized as its name has at least one non-capital letter.

Using S-diagram, it is possible to specify the relationships between two classes -- the subclass connection. The subclass connection is represented by the symbol --S-->; it implies inheritance from superclasses; that is, characteristics of superclasses will be reflected in their subclasses. Inheritance in S-diagram is in hierarchical form.

An attribute in S-diagram is represented by an arrow, which connects a domain class to a range class, accompanied by its name. An attribute may assign to a member of the domain class zero, one, or many members of the range class; likewise, each member of the range class may reference to

zero, one, or many members of the domain class via an attribute. These entity associations are called the cardinality of an attribute.

The cardinality of an attribute describes the functionality between the domain and range classes. Let

A = attribute

K1 = domain class

k2 = range class

m1 = minimum number of members of K1 referenced from a member of K2 via A

m2 = maximum number of members of K1 referenced from a member of K2 via A

n1 = minimum number of members of K2 connected to a member of K1 via A

n2 = maximum number of members of K2 connected to a member of K1 via A

The cardinality of A, combinations of n1 and n2, is tabulated in Table 4.1; whereas, the cardinality of the converse of A, A^{-1} is shown in Table 4.2. The "*" in the tables carries the meaning "many".

n1	n2	label combination
0	*	multivalued, optional
0	1	optional
1	1	(no label)
1	*	multivalued

Table 4.1 Cardinality of an Attribute

m1	m2	label combination
0	*	(no label)
0	1	unique
1	1	unique, onto
1	*	onto

Table 4.2 Cardinality of the Converse of an Attribute

The labels in Table 4.1 and Table 4.2 are represented graphically in an S-diagram. Fig 4.2 shows the graphical symbols of the labels.

Finally, the S-diagram may also express X-dependencies, the union of general existence dependencies and general

Label	Graphical symbol
multivalued	
optional	
unique	
onto	

Fig. 4.2 Labels in S-diagram

functional dependencies (see [EiRa85] for details). Furthermore, the semantics of S-diagram can be expressed in terms of a set of X-dependencies (also described in [EiRa85]).

4.2 Examples of Using an S-diagram

In this section, we will give two examples showing how to describe classes and attributes using an S-diagram and explaining the semantics of them. The first example is to describe a LISP class while the second is a registration application.

4.2.1 Example One -- LISP Class Definition

In this example, we want to define a class which is a LISP type; that is, each of its members is a LISP program. The S-diagram, S1, of this example is shown in Fig. 4.3.

Four classes are defined in S1, two lexicals and two non-lexicals. Class LISP-program is defined as having only one attribute, namely construct. Class lisp-type is a subset of LIST, which in turn is a subset of SEXPR. Therefore lisp-type inherits the characteristics of these two lexical classes.

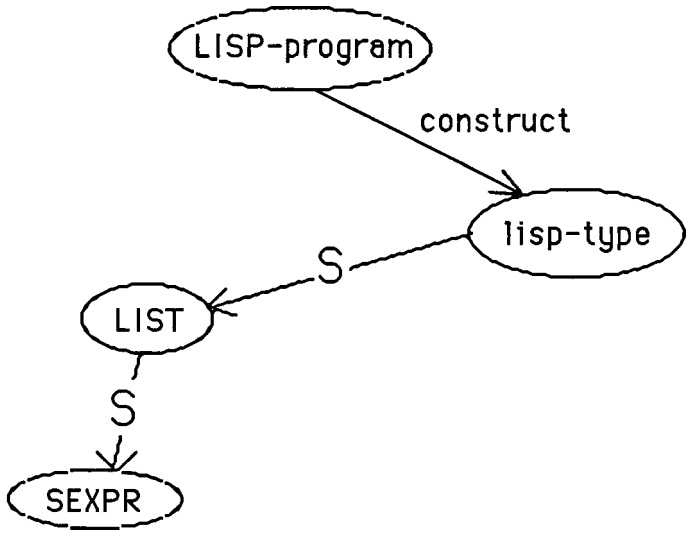

Fig. 4.3 S-diagram S1

Since the attribute construct does not have the label multivalued, a LISP-program must belong to no more than one lisp-type. However, the label optional is not there too; therefore, all LISP-programs must belong to one and only one lisp-type. Finally, the non-existence of labels onto and unique implies that the knowledge base can contain one or more LISP programs which have similar constructs.

4.2.2 Example Two -- Registration Application

In this application (see Fig. 4.4), INTEGER and TEXT are primitive data types while Student, Graduate-student, Teacher, Course, and Classroom are not. Graduate-student is a subset of Student. In the following, we will only explain the semantics carried by the attributes advised-by and enrolled-in. Then we will discuss some integrity rules which are not shown in the S-diagram S2.

The attribute advised-by connects Graduate-student to Teacher, and the label optional tells us that n1=0, n2=1, m1=0, and m2=*. Its semantics are described as follows: (1) a teacher can advise zero to many students; (2) if a graduate student is on non-thesis option, he does not need an advisor.

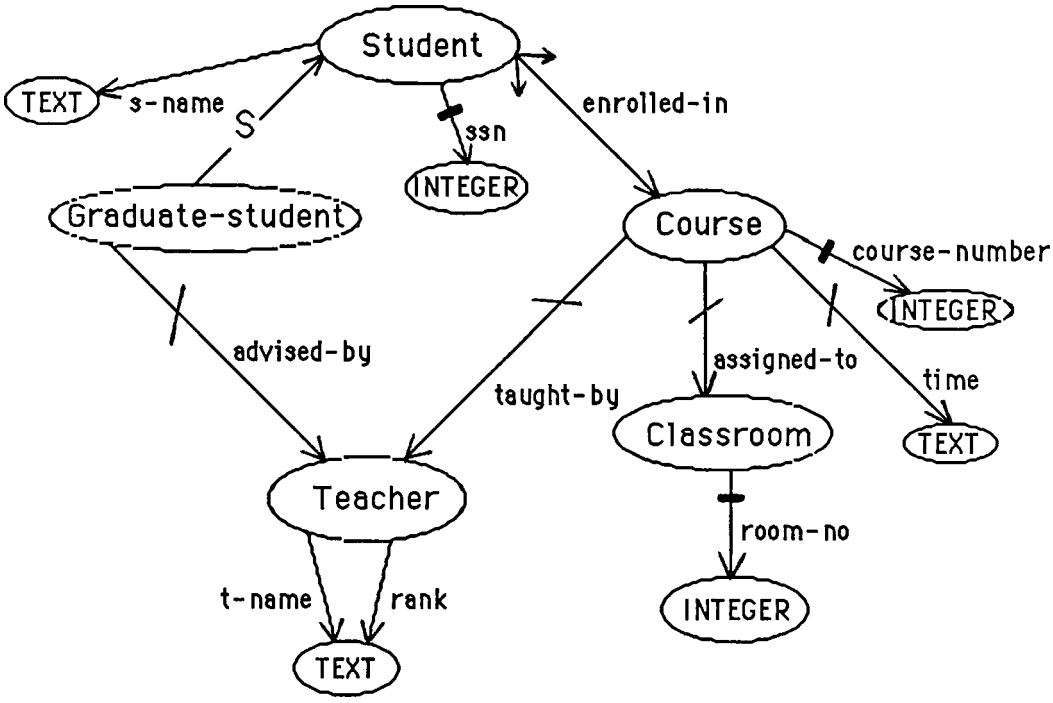

Fig. 4.4 S-diagram S2

Next, the attribute enrolled-in, connecting Student to Course, has the label multivalued. Its semantics is that a student can enroll in more than one course.

The labels optional, onto, and unique which do not appear in the attribute also imply some semantics:

1. A course will not be open if no one enrolls in it (because of the absence of the labels onto and unique).
2. A student must enroll in at least one course (because the label optional is not there).

Some integrity rules are not shown in S2. For example, a teacher cannot teach two or more courses at the same scheduled time. Similarly, a room cannot be assigned to two courses during the same period of time. These integrity rules can be expressed by introducing a new label to S-diagram. However, the discussion of this label is outside the scope of this thesis.

4.3 The Knowledge Base Manipulation Language

As mentioned in Chapter 2, data models provide a formal basis for tools such as DDL, DML, and QL that are implemented by a DBMS. In this section, we will discuss the Knowledge Base Manipulation Language in the DALI system.

The KBML is composed of the following functions: $KB-LOAD, $KB-UNLOAD, $KB-CREATE, $KB-DELETE, $KB-CONNECT, $KB-DISCONNECT, $KB-RETRIEVE, $KB-GET, $KB-FETCH, $KB-REPLACE, $KB-ADD-ATTR, $KB-DEL-ATTR, $KB-REWIND, $KB-READ, $KB-PRINT, $KB-BELONGS-TO. A KBML-function can be nested inside another, and virtually there is no limitation on the number of levels of nesting. In the following paragraphs, the syntax of this language will be examined (see Appendix II for the BNF of the KBML) and examples will be given for illustration.

The KBML-function $KB-LOAD loads the KB, specified in the argument, as well as the associated KBML program into memory; whereas, $KB-UNLOAD has the opposite effect of $KB-LOAD. Their formats are respectively.

($KB-LOAD <kb-name>)

($KB-UNLOAD <kb-name>)

The KBML-function $KB-CREATE creates a new entity and inserts membership to a class. It takes two arguments: the first one is the name of the class to which the new entity belongs while the second one is an association list which contains the attribute value pairs. Its format is

($KB-CREATE <class-name> <a-list>)

e.g.

```
($KB-CREATE 'STUDENT '((s-name Miller)
      (ssn 123456789)))
```

will create an entity of STUDENT with attributes s-name equal to "Miller" and ssn equal to "123456789". When the "create" function is successfully finished, it returns an entity number. The entity number functions as an identification of the entity; it will not be changed during the life of the KB.

$KB-CONNECT appends the additional attributes (attributes of subclass) to an already existing entity and inserts membership to a class. The function is of three arguments: the first one is an entity number, the second is a class name, and the last one is an association list that specifies additional properties of the entity. Its format is

```
($KB-CONNECT <entity-no> <class-name>
                <additional-attributes>)
```

e.g.

```
($KB-CONNECT 2 'GRADUATE-STUDENT '((advised-by 5)))
```

will connect the additional attribute "advised-by" (from GRADUATE-STUDENT) to the entity 2 (superclass of GRADUATE-STUDENT, perhaps STUDENT) and insert membership 2 to GRADUATE-STUDENT. It returns the entity number if the

connection succeeds.

$KB-DISCONNECT releases the attributes of a class specified in its arguments and deletes the membership from the indicated class.

($KB-DISCONNECT <entity-no> <class-name>)

e.g.

($KB-DISCONNECT 2 'GRADUATE-STUDENT)

will discharge all the attributes of GRADUATE-STUDENT in entity 2 and delete membership 2 from GRADUATE-STUDENT. When disconnection is done, the entity number will be returned.

$KB-DELETE ends the life of an entity and removes membership from all classes that it belongs to. $KB-DELETE returns the entity number and its format is

($KB-DELETE <entity-no>)

$KB-BELONGS-TO checks an entity number or a value whether it belongs to a class or a simple value set respectively. It returns a T if true otherwise a NIL. Its format is

($KB-BELONGS-TO <entity-no> <class-name>)

or

($KB-BELONGS-TO <value> <svs-name>)

The $KB-RETRIEVE function returns a set of entity numbers. These entity numbers represent all the entities in a particular class that satisfy the predicate specified in the retrieve function. Its format is

($KB-RETRIEVE <class-name> [<selection-criteria>])

where <selection-criteria> is the predicate to be satisfied. Mostly the selection criteria are involved with pattern matching language (see Chapter 5 for details).

Examples:

1. ($KB-RETRIEVE 'STUDENT '((s-name (Steve Miller))))

returns a set of entity numbers, possibly empty, from class STUDENT whose attribute s-name has the exact value of "Steve Miller".

2. ($KB-RETRIEVE 'STUDENT '((s-name ($ Miller))
(born-in Texas)))

retrieves all the students who were born in Texas with the last name equal to "Miller". Note that the "$" is used as a place holder in pattern matching; it is explained in detail in Chapter 5.

3. ($KB-RETRIEVE 'STUDENT '((s-name $X $X)))

retrieves all the students whose last names are also their

first names.

```
4.  ($KB-RETRIEVE 'TEACHER '((rank ($ professor))
                             (salary (#@ (greaterp ## 30000)))))
```

returns a set of entity numbers whose rank is either assistant professor or associate professor and whose salary is over $30,000 a year.

```
5.  ($KB-RETRIEVE 'GRADUATE-STUDENT
        '((advised-by (#@ (car ($KB-RETRIEVE 'TEACHER
                '((t-name (Christoph Eick)))))))))
```

retrieves all graduate students whose advisor is Christoph Eick.

The function $KB-GET retrieves the attribute values of an entity of a class. It takes as arguments an entity number and a list of selected attribute names. Its format is

```
($KB-GET <entity-no> <attributes>)
```

e.g.

```
($KB-GET 2 '(s-name ssn))
```

will return the values of s-name and ssn of entity 2 in the form of an association list, e.g. ((name (David Lee)) (ssn 123456789)).

$KB-RETRIEVE and $KB-GET can be combined into one step as $KB-FETCH. $KB-FETCH returns a list of association

lists. For example,

```
(((name (David Lee)) (ssn 123456789))
 ((name (John Smith)) (ssn 987654321)))
```

Its format is

```
($KB-FETCH <class-name> <attributes>
                  [<selection-criteria>])
```

$KB-REPLACE modifies attribute values of an entity. The old attribute-value pairs will be returned. Its format is

```
($KB-REPLACE <entity-no> <new-attribute-value-pairs>)
```

e.g.

```
($KB-REPLACE 2 '((ssn 420538928) (name (John Lee))))
```

will replace the old attribute values of ssn and name by 420538928 and (John Lee) respectively.

$KB-ADD-ATTR appends a value to an attribute. However, if the attribute has the property optional and does not exist in the entity, $KB-ADD-ATTR will insert the attribute along with the value to it. The return value is the entity number. The format of $KB-ADD-ATTR is

```
($KB-ADD-ATTR <entity-no> <attribute-name>
                  <attribute-value>)
```

$KB-DEL-ATTR does the reverse action of $KB-ADD-ATTR. When the last value in an attribute is deleted, the attribute name will be removed as well. $KB-DEL-ATTR returns the entity number. The format of this function is

```
($KB-DEL-ATTR <entity-no> <attribute-name>
                  <attribute-value>)
```

$KB-READ extracts and returns the attributes of a class, specified in its argument, from the entity indicated by the class-pointer. Class pointers are set to NIL when the KB is loaded. Therefore, before the read function can be executed, $KB-REWIND should be called to reset the class pointer to the top of a class. After the read function is executed, the class pointer will point to the next entity in that class. The format of $KB-REWIND and $KB-READ is respectively

```
($KB-REWIND <class-name>)

($KB-READ <class-name>)
```

Finally, the function $KB-PRINT works in the same way as $KB-READ except that return values will be directed to the system printer instead of the console. Its format is

```
($KB-PRINT <class-name> <attributes>)
```

4.4 An Example Program

In this section, we will give an example program using the KBML-functions in a LISP program. The following program will increase the salary of those employees who earn less than $1,000 a month when "flag" equals to 1. It prints out all the employees' name and salary when "flag" equals to 2.

```
(Defineq (example-program (flag)
  (let ((class 'EMPLOYEE)
        (new-salary nil))
    (cond
      ((equal flag 1)
        (for entity-no
          in ($KB-RETRIEVE class
               '((salary (#@ (lessp ## 1000)))))
         do (setq new-salary
                  (times 1.1
                     (cadr ($KB-GET entity-no '(salary)))))
             ($KB-REPLACE entity-no
                          `((salary ,new-salary)))))
      ((equal flag 2)
        (for entity-no
          in ($KB-RETRIEVE class)
         do (print
                  ($KB-GET entity-no '(name salary))))))))))
```

Chapter 5

Pattern Matcher

This chapter introduces the pattern matching language. The pattern matching language is used in the schema language to specify as a pattern the restrictions on the simple value sets. It is also used in the KBML as part of the selection criteria. The design and implementation of the pattern matcher are also discussed in this chapter.

5.1 The Pattern Matching Language

Pattern matching is the process of comparing two symbolic expressions to determine if one is similar to the other [Wins81, Wile84]. Though most pattern matching languages vary in the forms and expressive power, their concept or idea is more or less the same.

Pattern matching has been used in artificial intelligence for many years. Programs, for example, which deal with reasoning always need to access to knowledge about the world. This knowledge about the world might be expressed as pattern-like elements. The advantage of using pattern-like elements is that they can represent general

knowledge. For example, (cause (hit $x $y) (hurt $y)).

In this section, we will introduce the constructs of this language in detail (see Appendix I for the BNF of the pattern matching language) and examples will be given to help describe this sophisticated, though easy to understand, language.

In the rest of this chapter, we will assume a knowledge base with the following assertions:

```
(A B C D E)
(W (X Y) Z)
(P Q () R)
(L 3 M)
```

5.1.1 Identical Matchings

In its simplest case, a pattern can be any S-expressions. When this is the case, the match will be true only if the pattern and the assertion are exactly the same. For example, if we match the pattern

```
(A B C D E)
```

against the knowledge base, the result, in this case, is true. But it is false had the pattern changed to

(A B C D).

5.1.2 Place Holders

We increase the flexibility of our pattern matching language by adding two special symbols "$" and "*". Both of these symbols, indeed, serve as place holders and neither is bound to a value as a result of matching.

The first symbol "$" can match any atom or list in the corresponding position. As an example of this, let us consider the following pattern:

(W $ Z)

Since this pattern matches (W (X Y) Z) in the knowledge base, the match succeeds, but no value binds to this symbol.

The second symbol "*" works in the same fashion as the first one except that it can match zero, one or more positions in the assertion. For instance, matching

(P Q * R) or

(L 3 M *)

against the knowledge base will succeed, but matching

(* G)

will fail.

As we can see from the previous example, "*" in (L 3 M *) matches zero positions in the assertion (L 3 M) in the knowledge base. However, "$" is not capable of doing the same thing even though we have not given an example for it.

5.1.3 Pattern Matching Variables

In some cases, we would like to bind a value to a variable in its first occurrence so that this variable can be used as identical matching in the subsequent matches. Thus, pattern matching variables are included in this language.

A pattern matching variable is formed by attaching the special symbol "$" to a variable name. For instance, both

"$P" and "$ANY" are pattern matching variables while "$*" and "P$" are not.

When a pattern matching variable is encountered in the pattern, the system will first look for its binding value. If its binding value is found, the system will use this value for matching. On the other hand, if there is no binding value to this variable, the pattern matching variable will match like a place holder. If the match succeeds, the value of it will bind to this variable; otherwise not. Let us look at some examples:

Example 1

```
(A $VAR1 C D E)
```

When this pattern matches against the assertion (A B C D E) in the KB, the match succeeds and $VAR1 is bound to B, represented by ($VAR1 B).

Example 2

```
(A $X $X D E)
```

In this example, the pattern changes to a new form and the same assertion is being matched against. Since $X has no previous binding the first time, it will get the binding of

B. When $X is encountered the second time, the current binding of $X, which is B, will be used to match against C. The match fails at this point.

As mentioned earlier, knowledge about the world is often represented by pattern-like elements. Therefore, we might, sometimes, need to match a pattern against an S-expression that contains pattern matching variables too. But doing this may lead to some nasty problems unless we have some appropriate rules for binding variables. We adopt the unification rules that were explained in [Wile84], and we briefly restate them in here.

The unification rules state that when matching two items with variables in them, we first look for the binding value of that variable. If there is one, we will use it for continuous matching; otherwise, we will use the variable itself. However, when matching a variable against itself, we do not want to put that variable in the binding list because it will cause the searching of the value of that variable infinitely. To see why we do not want to do this, let us look at an example of this situation. Suppose we have

```
(A $X $X $X E)
```

against

(A $Y $Y $Y E)

When we match $X against $Y the first time, we bind $Y to $X since $X has no previous binding. Now when we match $X against $Y the second time, we will use the current binding of $X which is $Y. Since $Y does not have a current binding, we will bind $Y to $Y and put it into the binding list. When $X is matched against $Y the third time, we, again, get the current binding of $X, namely $Y, matched against $Y. But this time $Y has the binding $Y. So, an indefinite loop is created when the pattern matcher keeps on searching the current binding of $Y.

Another problem which the unification rules should deal with is called circularities. Circularities happens when the system attempts to match a variable against an item which contains the same variable. For example,

(A $X $X D)

against

(A $Y (C $Y) D)

In a situation like this, the pattern matcher will endlessly substitute $Y for (C $Y). Thus, the unification rules

should declare the match a failure.

5.1.4 Optional Occurrences

Our pattern matching language also observe the importance of optional occurences. An optional occurrence is denoted by a pair of curly brackets ({}). When an element is surrounded by the brackets, it means that this element can either exist in the assertion or not. For example, if a pattern contains an optional occurrence like the following:

(A {B} C)

the system should match

(A B C) or
(A C)

Note that only an atom or a list is allowed inside the brackets. Any special symbols or pattern matching variables will get an error message.

5.1.5 Restriction Functions

Restriction functions are implemented into the pattern matching language. They are indicated by a pound sign (#) followed by an at sign (@). A restriction function has the general format:

(#@ (<arbitrary LISP function> [<arguments>]))

<arbitrary LISP function> is any legitimate LISP functions. The system will pass this function to the interpreter without checking the meaning of it. Therefore if an arbitrary LISP function is not carefully chosen, it might cause run time errors or unexpected effects.

<arguments> is optional because some LISP functions may take zero arguments. If a LISP function takes one or more arguments, one of them must be "##". "##" is the special symbol we choose to denote the value of the corresponding position in the assertion.

Example 4

(L (#@ (NUMBERP ##)) M)

will succeed only if the first element is "L", the second

element is a number, and the last element must be "M".

Example 5

```
(L (#@ (NOT (NUMBERP ##))) M)
```

In contrast to the previous example, this one shows that the second element in the list must not be a number.

Example 6

```
(A B C (#@ (ATOMP ##)) E)
```

will succeed because (A B C D E) is in the knowledge base.

Example 7

```
(W (#@ (ATOM ##)) Z)
```

will fail when matching against the knowledge base because the second element in (W (X Y) Z) is a list.

Example 8

```
(P Q (#@ (LISTP ##)) R)
```

will match (P Q () R) in the knowledge base.

5.1.6 Permutation Functions

Sometimes we may want to match an assertion with certain elements in it; however, they need not necessarily be in any particular orders. For example, (a b c) or (c b a) are both acceptable. To reflect this need, we provide a function called #PERM that enables us to match the arbitrary permutation of elements E1, E2, . . . , En. Suppose we specify a pattern:

```
(A (#PERM B C D) K R)
```

The pattern will successfully match any of the following S-expressions:

```
(A B C D K R)
(A B D C K R)
(A C B D K R)
(A C D B K R)
(A D B C K R)
(A D C B K R)
```

5.1.7 Regular Expression Functions

In our pattern matching language, we also provide

regular expression functions. They are indicated by special symbols "#*", "#+", "#/", and "#&. The general format for regular expression functions is:

(<operator> <expression>)

<operator> can be one of the following:

#*	() (<expression>) (<expression> <expression>) (<expression> <expression> <expression>). . .
#+	(<expression>) (<expression> <expression>) (<expression> <expression> <expression>). . .
#&	repetition of <expression>
#/	any one of the occurrences in <expression>

While "#*", "#+", and "#&" are of one argument, "#/" can take more than one.

Examples:

(1) (#* (A))

will match the pattern (()) ((A)) ((A) (A)) ((A) (A) (A)) ...

(2) (#+ A)

will match the pattern (A) (A A) (A A A) ...

(3) (#& (A))

will match the pattern (A) (A) (A) ...

(4) (#& (A (#& (#@ (LISTP ##))) C))

will repeatedly match the successive lists with the first element in a list equal to A, followed by one or more lists, and then an atom C. Note that this example shows the recursive call to #& and it is the only regular expression function which allows recursive calls.

(5) (#/ A B C)

will match either A or B or C.

5.2 The Design and Implementation of the Pattern Matcher

The pattern matcher is composed of several functions. Each function is an implementation of the patterns described in section 1. The pattern matcher contains two parameters: pattern1 and pattern2. Pattern1 is the match expression

which may contain any combination of match functions and match variables; pattern2 is the S-expression to be matched. The pattern matcher scans the expression in pattern1 so as to find out which function should be called to match the S-expression. For example, when the special symbol "@" is detected following the symbol "#", the pattern matcher will call the module which is responsible for restriction function matching.

Most of the pattern matcher functions are implemented using recursive functions. An association list is used as an internal data structure to store the binding, variable-value pairs. In fact, this list is also the return value of the pattern matcher. When a NIL is returned, it means pattern1 and pattern2 are not a match. However, when a non-empty list is returned, it means the match is successful. If the list returned is (NIL), it is interpreted as a successful match without bindings; otherwise, the matching variable and their binding values are represented by an association list, e.g., (((\$A B) (\$X 1))).

Chapter 6

Schema Compiler

In this chapter we will discuss the syntax of the schema language (see Appendix III for the BNF of the schema language) and examine the data structure that is employed for the data dictionary. We will also discuss the design and implementation of the schema compiler. Before we get to these topics, we would like to, first, discuss the restrictions that are imposed on the schema language.

6.1 Restrictions on the Schema Language

We enforce some restrictions, for perusal purposes, on the schema language. These restrictions are explained in the following.

* Every keyword must be in small letters.
* An identifier (such as data class names, attribute names, etc.) must begin with a letter followed by zero or more characters. It can be of any length but not zero, and special symbols except the hyphen are not allowed.
* Simple value sets can be declared in any arbitrary order and so can data classes. However, all the simple value

sets have to be defined prior to any of the data classes definition.

* The schema is in free format, i.e., the user can start a line at any column. But, for readability it is suggested that the user indent a line properly.

6.2 Syntax and Semantic Rules of the Schema Language

A schema contains a line of identification followed by zero or more simple value set declarations and one or more data class definitions. In the following paragraphs, we will look at the syntax of schema identification, simple value sets, and data classes.

6.2.1 Schema Identification

The first line of a schema gives an identification. Its syntax is a simple one.

schema <SCHEMA-NAME>

The first word "schema" is used as a keyword while <SCHEMA-NAME> is a user-defined name.

6.2.2 Simple Value Set Declarations

Simple value sets are equivalent to data types in high level programming languages. They are established by being defined as a subset of the system predefined simple value sets: INTEGER, REAL, ATOM, LIST, and SEXPR. These predefined sets need not necessarily be declared in the schema. For example, the simple value set POS-INTEGER can be derived from the simple value set INTEGER by specifying a pattern which restricts values of it to positive integers. Simple value sets established from system predefined sets can serve as supersets for further derivations.

The syntax of a simple value set definition has three lines.

```
simple value set <SVS-NAME>
  subset of <SUPERSET-NAME>
    where <PATTERN>
```

or

```
simple value set <SVS-NAME>
  subset of <SUPERSET-NAME>
    where instances are <LIST>
```

The first line is composed of the keywords "simple value set" followed by an identifier, the name of a simple value set. The second line specifies the superset of the currently defined simple value set. We use the keywords

"subset of" followed by an identifier to denote this. Finally the "where" clause is expressed in the last line. The "where" clause can be either a pattern or a list of instances. If it is a pattern, it must be a list containing patterns recognized by the pattern matcher. Otherwise, it should be a list of enumerating atoms following the keywords "where instances are". Some examples of simple value set declarations are given in Fig. 6.1.

```
simple value set COLOR
  subset of LIST
    where instances are (red orange yellow green
                         blue indigo purple)

simple value set PRIMARY-COLOR
  subset of COLOR
    where instances are (red blue green)

simple value set POS-INTEGER
  subset of INTEGER
    where (#@ (GREATERP ## 0))
```

Fig. 6.1 Examples of Simple Value Sets

The first simple value set COLOR is declared as a list of seven elements. Its superset is LIST, which is a subset of SEXPR. Both LIST and SEXPR are predefined in the system; therefore, they do not need to be declared as a simple value set in the schema.

The second simple value set is called PRIMARY-COLOR. It restricts itself to red, blue, and green out of simple value set COLOR. As mentioned above, the order of these two simple value sets is not important because the schema compiler can take care of forward references.

The last simple value set is not in the same form as the previous two. It employs the second form of the "where" clause, which is a pattern. The simple value set POS-INTEGER specifies that the number has to be a positive integer number. The special symbol ## is used as a place holder in pattern matching. The real values will be substituted into these places during the run time.

6.2.3 Data Class Definitions

The syntax of data class definitions is more complicated than that of the simple value sets. Like the simple value set, a data class might be a subset of other class. Furthermore, it might overlap with some other classes. The structure of data class definition is shown in Fig. 6.2.

```
data class <CLASS-NAME>
  subset of <SUPERCLASS-NAME>
    overlaps with <OTHER-CLASS>

  simple attributes:
    <ATTRIBUTE-NAME>
      property: <PROPERTY-VALUES>
      default: <VALUE>
      constraints: <PATTERN>
      type: <SIMPLE-VALUE-SET-NAME>
    <ATTRIBUTE-NAME>
        .
        .
        .

  role attributes:
    <ATTRIBUTE-NAME>
      property: <PROPERTY-VALUES>
      default: <VALUE>
      constraints: <PATTERN>
      type: <CLASS-NAME>
    <ATTRIBUTE-NAME>
        .
        .
        .

  entity local constraints: <CONSTRAINT-FUNCTION>

  general constraints: <CONSTRAINT-FUNCTION>

  predefined operations: <PRIMITIVE-OPERATIONS>
```

Fig. 6.2 Structure of Data Class

The order of the keywords except those (property, default, constraints, and type) used to describe attributes (see Fig. 6.2) is very important because the schema compiler scans the text using the above order. Although Fig 6.2 shows all the keywords that a data class could possibly have, a data class does not necessarily have to have all these specifications. If, for example, a data class does not overlap with other class, the line "overlaps with" will not appear in the definition.

Types in simple attributes are expected to be simple value sets; whereas, types in role attributes must be any of the data class names defined inside the schema. <PROPERTY-VALUES> are values defined in the S-diagram. They can be any combination of these: unique, optional, multivalued, and onto. <VALUE> is the default value to be used if the attribute-value pair is omitted in the argument when an entity is instantiated. Finally <PATTERN> is a pattern to be satisfied by the attribute.

Entity local constraints are constraints that apply to entities in a particular class only. They have to be a boolean LISP expression -- either a T or a NIL is returned.

General constraints are arbitrary LISP expressions, and

they apply to a class instead of entities.

Finally, predefined operations are data manipulation functions that allow the user to manipulate entities in a particular class. Data manipulation functions are denoted by the prefix "$KB-". Functions that are currently implemented in the system include: $KB-CREATE, $KB-DELETE, $KB-CONNECT, $KB-DISCONNECT, $KB-RETRIEVE, $KB-FETCH, $KB-GET, $KB-REPLACE, $KB-ADD-ATTR, $KB-DEL-ATTR, $KB-REWIND, $KB-READ, and $KB-PRINT. These are considered to be the primitive operations in the system. If this line is omitted, all operations will be assumed.

6.3 Data Structures of the Data Dictionary

A data dictionary is actually a small data base which contains all the necessary information that is required to generate a KBML program. This information may be called intermediate codes.

In LISP, a property list can be used for storing information because it is easy to deposit or retrieve information to or from it via LISP functions DEFPROP and GET. Nonetheless, a property list itself is not enough to be a candidate because it can store property value only.

Fortunately, with some modifications a property list can be generalized to a frame. A frame has room to specify more than property values, for example, default values, messages, computed values, and inherited values.

One way to define a frame [Wins81] is as a nested association list. On the highest level is the frame name and each sub-level is a nested association list. To demonstrate the use of a frame as a data dictionary, we give the following example.

Example

```
(REGISTRATION
    (simple-value-set TEXT)
    (TEXT
       (pattern (#@ (LITATOM ##))))
    (data-class STUDENT GRADUATE-STUDENT TEACHER)
    (STUDENT
       (superset-of GRADUATE-STUDENT)
       (attribute-names sname ssn)
       (sname
          (category simple)
          (type TEXT))
       (ssn
          (category simple)
          (type INTEGER))
       (operations $KB-CREATE $KB-RETRIEVE))
    (GRADUATE-STUDENT
       (subset-of STUDENT)
       (attribute-names advised-by)
       (advised-by
          (category role)
          (type TEACHER)
          (property optional)))
    (TEACHER
       (attribute-of GRADUATE-STUDENT)
       (attribute-names rank)
```

```
(rank
   (category simple)
   (type TEXT))
   (default assistant professor)
(general-constraints
    (LESSP ($KB-RETRIEVE SELF) 10)))
```

In this example, REGISTRATION is the frame name. It contains three sub-levels. Basically, the first levels store the names of simple value sets and data classes; the second levels store superset names, subset names, attribute names, general constraints, local entity constraints, and predefined operations; and the third levels store attribute properties such as data types, default values, attribute constraints, and cardinality of attributes.

The data dictionary is implemented using abstract data types. The operators associated with it are $DD-GET, $DD-INHERIT, $DD-INHERIT-SVS, $$DD-BACK-INHERIT, $DD-PUT, $DD-DELETE, and $DD-APPEND. $DD-GET gets a value from a slot; whereas, $DD-PUT deposits a value to it. $DD-INHERIT not only gets a value from a specified slot, but also inherits values from its superclasses. $DD-INHERIT-SVS is a variation of $DD-INHERIT since it is specialized in inheriting simple value sets only. $DD-BACK-INHERIT virtually does the opposite task of $DD-INHERIT. Instead of inheriting its superclasses, it looks for subclasses as well as classes which overlap with it. $DD-DELETE removes a slot

from a data dictionary entry. Lastly, $DD-APPEND appends a value to a slot. If the slot is not found, it will create one before the value is appended to it.

6.4 The Design of the Schema Compiler

The schema compiler uses a one pass technique with three phases grouped into it. The first phase is lexical analysis, the second phase is syntax analysis, and the third phase is KBMF-code generation. The operations of these phases are interleaved, with control alternating among them.

Error handling interacts with all three phases. When an error in the source program is detected during the first two phases, it is reported to the error routine; however, the schema compiler will not terminate there because it attempts to detect as many errors as possible in one compilation.

6.4.1 Lexical Analysis

In the first phase, the lexical analyzer or scanner is called. The input of the lexical analyzer is, of course, the conceptual schema while the output of it is a stream of tokens.

A token is a group of characters that logically belong together [AHU 78]. What is called a token depends on the language at hand. In most languages, a constant, an identifier, an operator symbol, a keyword, and a punctuation symbol are treated as a token. In the schema language we are dealing with, there is no operator symbol. However, we have something called "list" that most languages do not have. A list is also called a token because it is treated as a singly logical entity in LISP. In summary, the lexical analyzer will recognize the following as tokens:

1. constants -- Either integers or real numbers.
   e.g., 1, 200, 3.46
2. identifiers -- Begin with a letter followed by zero or more characters.
   e.g., ID1, EMPLOYEE
3. lists -- Elements surrounded by parentheses. Elements can be either lists or word-like objects called atoms [Wins81]. When an element is a list, it forms an hierarchical structure.
   e.g., (a b c), (the following is a list (1 2 3))
4. keywords -- All keywords are in small letters The following are named as keywords in the schema language: schema, simple, value, set, subset, of, where, instances, are, data, class, overlaps, with,

attributes, type, default, property, constraints, role, entity, local, general, predefined, and operations. Note that these keywords are not reserved; the user can use them as identifiers.

5. punctuation symbols -- Only two punctuation symbols are used in the schema language. They are comma and colon.
6. special symbols -- Special symbols are mainly used in the pattern matching language. These symbols are: #@, ##, #PERM, #*, #+, #&, and #/.
7. property values -- The values of attribute property include: unique, optional, onto, and multivalued.
8. predefined operations -- Thirteen operations are currently implemented: $KB-CREATE, $KB-DELETE, $KB-CONNECT, $KB-DISCONNECT, $KB-RETRIEVE, $KB-FETCH, $KB-REPLACE, $KB-GET, $KB-ADD-ATTR, $KB-DEL-ATTR, $KB-REWIND, $KB-READ, and $KB-PRINT.

The lexical analyzer interacts with four functions: $SCAN-A-NUMBER, $SCAN-A-WORD, $SCAN-A-LIST, and $SCAN-A-PUNCTUATION. It always reads one character ahead. Therefore, a proper function would be called based on the current character. For example, if the current character is a number, it will call the function $SCAN-A-NUMBER; a letter or special symbol "$" or "#" will invoke $SCAN-A-WORD; a right-parenthesis will invoke $SCAN-A-LIST; and a comma or

colon will invoke $SCAN-A-PUNCTUATION.

The lexical analyzer is responsible for producing a source listing and identifying any illegal input characters. It can also filter out unnecessary blanks embedded in the text.

The lexical analyzer is not called to produce the entire sequence of tokens on an intermediate file. Rather, it is called as a function by the syntax analyzer each time a new token is desired.

6.4.2 Syntax Analysis

The second phase of the schema compiler is called syntax analysis. Its input is the output of the lexical analyzer. The syntax analyzer checks the pattern of input whether or not it matches the specification for the source program.

The syntax analyzer is composed of three functions: $CHK-SCHEMA-NAME, $CHK-SVS, and $CHK-DATA-CLASS. $CHK-SCHEMA-NAME is first called to check the first line of the schema. Once it is done, it will give control to $CHK-SVS to analyze the syntax of all the simple value sets.

When it is finished, the last function $CHK-DATA-CLASS should be called to do syntax analysis for all data classes.

Since the complexity of the schema language is small, the parsing techniques are not necessary at this stage. The syntax analyzer uses a simple scheme; that is, it analyzes the contexts of a conceptual schema word by word. Instead of building a parse tree in this phase, the syntax analyzer builds an entry in the data dictionary. This entry contains the intermediate codes for the currently compiled schema. When a source line is detected error free in the first two phases, the intermediate codes of this line will be inserted into the entry, using the data dictionary operators. Each entry in the data dictionary is unique; therefore, before the first phase begins, the schema compiler checks if the data dictionary entry has already been created. If so, it will prompt the users whether the old entry should be overwritten. If the answer is no, the compilation is aborted.

If anything unexpected happens during syntax analysis, the error routine is invoked for error handling and the error flag is incremented by one. When a flaw is found in a keyword during syntax analysis, the rest of the line will be skipped because there is no way to tell what exactly the

keyword is and it makes no sense to scan the rest of the line. The scanner then tries to return a possible keyword following the erroneous line, and the syntax analysis resumes from there.

When the second phase is finished, the schema compiler will check the value of the error flag. If the value is greater than zero, it will call $DD-DELETE to remove the entry from the data dictionary and compilation is aborted; otherwise, the third phase begins.

6.4.3 KBMF-code Generation

The KBMF-code generator produces a set of KBML-functions which will be used by the end users for accessing the knowledge base. It takes as input the data dictionary and generates as output a program containing all the KBML-functions which were specified as predefined operations in the conceptual schema. KBML-functions generated can be compiled so that they can run quicker than the symbolic codes. Before the design and implementation of the KBMF-code generator is discussed, we would like to describe the context of a KBML program.

6.4.3.1 An Overview of KBML Program

Fig. 6.3 shows the layout of a KBML program. Basically a KBML program contains several KBML-functions. The number of functions produced depends on the number of allowed operations specified in the conceptual schema. Suppose, a conceptual schema defines four classes, HUMAN-BEINGS, PATIENT, HOSPITAL, and TREATMENT. The permissible operations for HUMAN-BEINGS are $KB-CREATE, $KB-DELETE, and $KB-RETRIEVE; for PATIENT $KB-CONNECT, $KB-DISCONNECT, $KB-RETRIEVE, and $KB-READ; for HOSPITAL $KB-CREATE and $KB-DELETE; and for TREATMENT $KB-CREATE, $KB-DELETE, and $KB-READ. When this conceptual schema is compiled, the syntax analyzer will transform all these predefined permissible operations for each class into intermediate codes and store them in the data dictionary. Next, when the program generator is invoked, it will generate, in reference to the data dictionary, a program with six functions in it, namely, $KB-CREATE, $KB-DELETE, $KB-CONNECT, $KB-DISCONNECT, $KB-READ, and $KB-RETRIEVE.

```
(Defineq (<kbml-function> (<class-name> ...)
   cond ((not (atom <class-name>))
         ($error code)
         nil)
           .
           .
           .
         (t (cond ((equal <class-name> <value>)
                     .
                     .
                     .
                                        )
                  ((equal <class-name> <value>)
                     .
                     .
                     .
                                        )
                  (T ($error code)))
            (cond ((null error)
                   (<dbml-function>
                           <class-name> ...))
                   (t ($error code)))))

(Defineq (<kbml-function> (<class-name> ...)
              .
              .
              .
                                      ))
```

Fig. 6.3 Layout of KBML Program

Although the contexts of these functions are different from each other, they do have few things in common. First, in the outset of each function, all the parameter values will be tested. For instance, a class name have to be an atom and a criteria must be an association list. Second, a conditional clause is used to test the names of all the permissible data classes. In the last example, the permissible data classes for $KB-CREATE are HUMAN-BEINGS, TREATMENT, and HOSPITAL only. Creating entities for PATIENT will cause an error message. Similarly, the permissible data classes for $KB-RETRIEVE are HUMAN-BEINGS and PATIENT; for $KB-DELETE HUMAN-BEINGS, HOSPITAL, and TREATMENT; for $KB-CONNECT and $KB-DISCONNECT PATIENT only; and for $KB-READ TREATMENT and PATIENT. Third, an error routine is called whenever a flaw or violation is detected. Fourth, their corresponding low level function will be called upon to manipulate the physical data of the KB when all the validations are done and no errors have been found.

On the other hand, the differences of the contexts of these KBML-functions are explained in the following paragraphs.

The contexts of $KB-CREATE and $KB-CONNECT are most complicated. Before checking the redundant attribute value

pairs, they insert default values for the missing arguments. Then they go on checking the optional attributes, the attribute types and constraints, the cardinality of attributes between the domain and range classes, the general constraints, and the entity local constraints. For $KB-CONNECT, it also checks whether the entity number is allowed for the connection of the additional attributes.

The contexts of $KB-DELETE and $KB-DISCONNECT look more or less the same. They insure the cardinality of attributes is not violated when an entity is deleted from all classes or disconnected from an entity.

For $KB-RETRIEVE, it needs to verify the attribute names in the selection criteria. Likewise, $KB-FETCH checks the same things as $KB-RETRIEVE plus the validation of the selected attribute names.

The contexts of $KB-REPLACE, $KB-ADD-ATTR, and $KB-DEL-ATTR not only verify attribute names, but also perform data integrity tests. When $KB-DEL-ATTR deletes the last attribute value, it may cause consistency violation if that attribute is not optional. Similarly, when $KB-ADD-ATTR puts an additional attribute value to a non-multivalued attribute, the operation should be rejected.

For $KB-REPLACE, it must insure that the new values will not violate any of the consistency rules.

Functions like $KB-GET, $KB-REPLACE, $KB-ADD-ATTR, and $KB-DEL-ATTR do not carry a class name in their parameter. Therefore, their class name and attribute names can only be verified from the KB during the run time.

Finally, $KB-REWIND, $KB-READ, and $KB-PRINT have the simplest contexts. All of them have no attributes to verify; however, $KB-READ and $KB-PRINT need to pass the attribute names of the requested class to their corresponding low level function.

All of these high-level functions generated by the code generator are machine-independent. In other words, things changed in the low-level functions would not affect the code generator program.

6.4.3.2 The Design of the KBMF-code Generator

The KBMF-code generator is composed of twelve functions along with the driver program. Each function except one is responsible for generating a KBML-function. For example, $GEN-KB-CREATE will generate $KB-CREATE. For $GEN-KB-RP, it

is responsible for generating two functions: $KB-READ and $KB-PRINT. The driver program opens a file with its name equal to the schema name to store the generated codes (KBML-functions). Then it will call each function in sequence to generate the codes if necessary.

The first thing for each function to do is to check with the data dictionary to see whether there is a need to generate this particular KBML-function. If not so, control will go back to the driver program and the next function is called.

In each function, templates are declared at the beginning of it. These templates are program segments; they contain some place-holders which will be substituted by the values retrieved from the data dictionary. A list called LINE is used to temporarily store the generated codes before they are written to the file opened by the driver program. The reason is that I/O frequency will be reduced; as a result, the program generator is more efficient. Finally, a stack is needed to temporarily store the right-parentheses because in some situations, when the left parenthesis of a list is appended to LINE but the list is not yet finished, the right parenthesis should be pushed onto the stack; and when it comes to the end of the list, the right parenthesis

will be popped out from the stack and appended to LINE.

6.4.4 Error Handling

$WRITE-ERRMSG, the error handler is invoked when a flaw is occurred in the source program during compilation. The error handler takes as a parameter an error code, and it stores the corresponding diagnostic message into a global list called ERRMSG. These error messages will not be output to the listing file until the end of the source line is reached. Despite the error occurs in the source line, the syntax analyzer will continue checking the rest of the schema in order to detect as many errors as it can in one compilation.

At the end of each source line, a routine named $SC-PRINT-ERR-MSG is invoked to check the contents of ERRMSG. If it is not null (or empty), its contents will be printed underneath the source line; otherwise, no action will be taken.

6.4.5 The Forward References Problem

The forward references problem is inevitable in a one-pass compilation. To explain what the forward

references problem is, consider the following example. Suppose in a schema specification, role attribute hosp-name is defined as with type equal to HOSPITAL, e.g.,

```
role attributes:
  hosp-name
    type: HOSPITAL
```

The problem arises here: if HOSPITAL, a data class name, is defined ahead of the current data class, the parser will accept it as a legitimate type; however, if HOSPITAL has not yet been defined, the parser will not have any knowledge, at that point of time, whether or not this data class is defined later in the schema. This creates the forward references problem.

The syntax analyzer may solve the problem by using a list to store all the unresolved names. Each time when a new data class is compiled, the data class name is checked against this list. If a match is found, the unresolved name is deleted from the list. So at the end of the compilation, the undefined data class names remain unresolved. These names will be output to the listing file along with the diagnostic error message.

Chapter 7

Storage Structure Program

In the last two chapters, we discussed two of the components of DALI -- the pattern matcher and the schema compiler. This chapter discusses the last component -- the storage structure program. First, the selection of data structure for the DALI-KB will be discussed. Next, the roles of the storage structure program are described. Then the design and implementation are discussed. Lastly, the features of the Xerox 1186 AI system in which DALI is implemented are mentioned.

7.1 Selection of Data Structure for DALI

In our KBMS, we should allow sequential access and key access. Sequential access is used when the whole class must be traversed. For example, $KB-RETRIEVE needs to traverse a class sequentially to find out which entities that satisfy the selection criteria. On the other hand, key access is used when an entity number is available and quick access is needed. Furthermore, key access is also used to check whether an entity number belongs to a certain class.

For sequential access, we choose to use a list as the data structure. This list stores all the memberships (entity numbers) of a class. Sequential mode is also involved in key access because the entity numbers in the sequential list will be used as the keys to access the contents of the entities.

For key access, we need to find some data structure which allows quick access with an entity number as a key. Some candidates are B-tree [AHU 83], hashing [Mehl84], and hash arrays (an INTERLISP-D data structure). Hash arrays provide a mechanism for associating arbitrary LISP objects ("hash keys") with other objects ("hash values") such that the hash value associated with a particular hash key can be quickly obtained.

In order to compare their performances, we conducted some empirical tests. The B-tree was implemented using the record package in INTERLISP-D. The record package is an abstract data type; it has operations like CREATE, FETCH, REPLACE, WITH, and TYPE?. In our program the non-leaf node was declared as

```
(Datatype Btree
       (Firstchild Lowofsecond Secondchild
           Lowofthird Third))
```

and the leaf node as

(Datatype Leaf23 (element23))

The B-tree program was implemented in ADT too. The operators associated with the B-tree were BTREE-CREATE, BTREE-INSERT, BTREE-DELETE, BTREE-RETRIEVE.

The hashing program was also implemented using ADT's. The collision problem was resolved by linear rehashing. The operators associated with hashing were HASH-CREATE, HASH-INSERT, HASH-DELETE, and HASH-RETRIEVE.

Since there are micro-coded functions for creating hash arrays, putting a hash key/value pair in a hash array, and quickly retrieving the hash value associated with a given hash key; the hash array program simply used these micro-coded functions to implement the operations: HARRAY-CREATE, HARRAY-INSERT, HARRAY-DELETE, and HARRAY-RETRIEVE.

The tests were executed in the manner described as follows and the results listed in Table 7.2.

1. Two thousand records were inserted into a 2-3 tree.
2. One thousand records were then deleted.

3. The elapsed time was recorded.
4. One thousand records were retrieved, but 50% of them were not in the 2-3 tree.
5. The time consumed was marked down.
6. A separate test was performed after the above operations.
7. One thousand records were retrieved from the 2-3 tree with no misses.
8. The elapsed time was marked down.
9. Procedures 1-8 were repeated with hashing and hash array.

The results of these tests are tabulated as follows:

	B-tree	Hashing	Hash Array
INSERT and DELETE	892441	39908	31722
RETRIEVE (50% not found)	81432	12332	36238
RETRIEVE (100% found)	81432	17624	5013

Table 7.2 Empirical Test Results

The empirical results show that when these three data structures run on the Xerox 1186 workstation, hash array is much faster than hashing and B-tree for inserting or deleting a record. It also outperforms its competition in data retrieving.

In summary, hash array was chosen as the data structure for our DALI-KB. The hash key is the entity number and the hash value is an association list. The first attribute-value pair of the association list will be used to store all the class names the entity belongs to.

7.2 An Overview of the Storage Structure Program

The storage structure program is a set of low-level functions which directly modify the physical data of the knowledge base. These functions are one-to-one correspondences to the high level KBML-functions. The purpose of having this layer is that when changes are made in this layer due to performance reasons (e.g. the data structure for the KB is changed), they would not affect anything in the user interface layer. Low level functions are indicated by using the prefix "$DB-".

All of these low-level functions are machine-dependent.

The users are not permitted to access these functions; as a result, they would not intentionally or unintentionally destroy the contents of the KB.

7.3 The Design of the Storage Structure Program

For each of the data manipulation functions, we will implement a function which knows the data structure of the KB. This low level function is responsible for performing the request from the high level function. When requests are performed, things are changed in the KB. For instance, entity numbers can be reused when entities are deleted. Therefore, some data structure is needed to keep track of the status of the KB when things are changed. We will call it the Knowledge Base Table. The Knowledge Base Table contains the information like the next available entity number, the recycled entity numbers, the memberships of each class, and the class pointers. The Knowledge Base Table is currently an association list. Fig. 7.1 shows an example of how the information is stored in the table.

```
((RECYCLED-ENTITY-NOS (2 14 1))
 (NEXT-ENTITY-NO 15)
 (MEMBERSHIP (TEACHER (3 6))
             (STUDENT (4 5 7 9 10))
             (GRADUATE (5 9))
             (COURSE (8 11 12 13)))
 (CLASS-POINTER (TEACHER NIL)
                (STUDENT 7)
                (GRADUATE NIL)
                (COURSE 8)))
```

Fig 7.1 An Example of the Knowledge Base Table

Since the low level functions work closely with the Knowledge Base Table, it is logically to design the table as an abstract data type. The operations in the Knowledge Base Table are $KBT-GET-MBRSHIP, $KBT-DELETE-MBRSHIP, $KBT-INSERT-MBRSHIP, $KBT-GET-ENTITY-NO, $KBT-GET-CLASS-PTR, $KBT-RECYCLE-ENTITY-NO, and $KBT-RESET-CLASS-PTR.

$KBT-GET-ENTITY-NO first searches for the recycle list. If it is not empty, the first element will be taken out and returned. Otherwise, the value of NEXT-ENTITY-NO will be returned and NEXT-ENTITY-NO is incremented by one.

$KBT-DELETE-MBRSHIP, $KBT-INSERT-MBRSHIP, and $KBT-GET-MBRSHIP all deal with attribute MEMBERSHIP. The first operator deletes the entity number from the class membership list specified in its argument, the second operator inserts a member to the list, and the last one returns the whole list.

$KBT-GET-CLASS-PTR returns the current class pointer; whereas, $KBT-RESET-CLASS-PTR rewinds the class pointer to the top of a class.

Finally, $KBT-RECYCLE-ENTITY-NO deposits an entity number to the recycle list.

With these operators, the storage structure programs can easily do the house-keeping work. For example, when an entity is deleted from the KB, $DB-DELETE sets the hash value to NIL, then it calls $KBT-DELETE-MBRSHIP to delete memberships from all classes and finally it calls $KBT-RECYCLE-ENTITY-NO to recycle the entity number.

7.3 Implementation Notes

DALI is currently installed in the Xerox 1186 AI system. The 1186 is a single user workstation. The central processor is implemented in Schottky TTL technology, based on a high-speed version of the Advanced Micro Devices 2901C bit slice processor with custom LSI and gate arrays are used for microinstruction latching and decoding and bus arbitration. In order to improve performance, an independent coprocessor (Intel 80186) is used for handling all I/O devices (except the display controller). Eight thousand hand-tuned microcode instructions for LISP and Prolog are installed in the writeable control store for maximizing the performance of an integrated symbolic programming environment.

Other features include a keyboard, a three-button optical mouse, a high resolution bit-mapped graphical display, an integral Ethernet II controller, a 1.6 MB main memory, a 80 MB local rigid disk, a 5 1/4" floppy disk drive (360 KB formatted), and two serial communications ports: an RS-232-C DTE communications port and an RS-232-C DCE printer port.

DALI is implemented using INTERLISP-D which amplifies

the power of LISP. For many years, various implementations of INTERLISP have been used for large, knowledge-based systems and advanced user-interfaces. It is specially designed to take advantage of the technology of the high-powered, single-user, networked workstations that it runs on. Moreover, windows, mouse input, graphics, and communications all fit neatly within the language.

The time for one 48-bit microinstruction executed from a writeable 8K word control store is one cycle (125 nanoseconds). The system uses a multitasking scheme where cycles are grouped into clicks, with three cycles in one click. During each click (375 ns), three microinstructions and one simultaneous memory reference is accomplished. On the average, a one-bye LISP instruction takes about 562 nanoseconds or at a rate of 1.77 MIPS.

The memory system provides exactly one access per click: the first cycle of a click sends an address, the second cycle delivers a word to be written, and the last cycle returns the word which has been read. Thus, a 32-bit data fetch is accomplished in 750 nanoseconds, yielding a memory bandwidth of 41 Mbits/sec.

Chapter 8

Summary and Conclusions

The first prototype of DALI has been implemented on the Xerox 1186 AI system. It contains a schema compiler, a pattern matcher, and a storage structure program. The schema compiler is slow at this stage. However, this is not a major problem because usually a schema is not compiled very often -- probably once every three months. The pattern matcher is very versatile. It not only covers a large variety of patterns, but also handles the combinations of them. The performances of the storage structure program are satisfactory since the data structure of the KB was selected via some empirical tests. Moreover, the storage structure program was implemented using abstract data types; therefore, when better data structure is found, the abstract data types are easily modified.

The architecture of DALI is multi-layered. This allows less changes to be made when it is installed on other systems. In a multi-layer architecture, only the lower level layer has to be modified. The changes in the lower level layer is inevitable because the internal data structure vary differently from machine to machine.

Besides, in the performance standpoint, it is too costly not to use machine-dependent codes. When DALI is installed on a system not using INTERLISP-D language, some macros have to be written to convert the differences between the two dialects. But in general, this is not so hard to accomplish. Furthermore, codes in the low level layer have to be rewritten.

DALI will be enhanced in the future. The future research works include the following:

1. Exception handling will be added to DALI to ensure the appropriate execution of operations. When any violations of rule constraints occur, the exception handler is called to recover from errors.
2. The on-line documentation and help files will be extended so that DALI becomes more user-friendly.
3. A graphical-oriented data modelling tool which will simplify knowledge base design will be implemented.
4. The schema language will be expanded so that user-defined operations can be defined by the language. User-defined operations are built upon the primitive KBML-functions.
5. Forward chaining and backward chaining techniques will be accompanied with pattern matching for accessing the KB. DALI should allow good performances for the

implementation of these algorithms.

6. Additional storage structures have to be provided for the internal representation of the KB. The knowledge base designer will have to choose which storage structure that is well suited for a particular application.
7. Some mechanisms will be implemented to handle rules and uncertain knowledge in the KB.
8. An inference engine will be used as the front-end driver.
9. A user view definition facility (VDF) will be added to DALI. This facility will allow the user to view a certain part of the KB as his own knowledge base.
10. A performance analysis tool for DALI will be implemented.
11. A security software will be developed to protect the KB from any unauthorized users.
12. Inverted files will be installed to the system.

These features are projected to be finished within two years and number 1, 4, and 7 have been already under exploration by other students in their master theses.

All in all, DALI is a user-friendly and easy-to-use knowledge base management system. It is designed for large

computerized knowledge bases; therefore, in the long run, it can cut down the maintenance cost, software overhead, and the length of the application programs. And because DALI supports object-oriented programming, it will help people with less AI knowledge write knowledge-based programs.

Appendix I

The BNF of the Pattern Matching Language

```
<pml> ::=
    (<patterns>)

<patterns> ::=
    <pattern>
   |<patterns> <pattern>

<pattern> ::=
    <s-expression>
   |<place-holder>
   |<pm-variable>
   |<optional-occurrence>
   |<pm-function>

<s-expression> ::=
    atom
   |list

<place-holder> ::=
    *
   |$

<pm-variable> ::=
    $letter{letter|digit}*

<optional-occurence> ::=
    {<s-expression>}

<pm-function> ::=
    <restriction-function>
   |<permutation-function>
   |<regular-expression-function>

<restriction-function> ::=
    (#@ <lisp-function>)

<lisp-function> ::=
    LISP function

<permutation-function> ::=
    (#PERM <permutation-list>)

<permutation-list> ::=
    <s-expressions>
```

```
<s-expressions> ::=
    <s-expression>
   |<s-expressions> <s-expression>

<regular-expression-function> ::=
    (<regular-expression-symbol> <s-expression>)

<regular-expression-symbol> ::=
    #* | #+ | #& | #\
```

Appendix II

The BNF of Schema Language

```
<schema-language> ::=
    schema <id> <schema-description>

<id> ::=
    identifier

<schema-description> ::=
    [<svs-list>] <data-class-list>

<svs-list> ::=
    <svs>
   |<svs-list> <svs>

<svs> ::=
    simple value set <id> <svs-option>

<svs-option> ::=
    subset of <id> where <restrictions>

<restrictions> ::=
    <pattern>
   |<instances>

<pattern> ::=
    <pml>   (see Appendix I)

<instances> ::=
    instances are <instance-list>

<instance-list> ::=
    list

<data-class-list> ::=
    <data-class>
   |<data-class-list> <data-class>

<data-class> ::=
    data class <id> [subset of <id>]
       [overlaps with <id>] <class-description>

<class-description> ::=
    <simple-attr-declaration> <role-attr-declaration>
          <class-attributes>
```

```
<simple-attr-declaration> ::=
    simple attributes: <simple-attr-list>

<simple-attr-list> ::=
    <simple-attr-desc>
   |<simple-attr-list> <simple-attr-desc>

<role-attr-declaration> ::=
    role attributes <role-attr-list>

<role-attr-list> ::=
    <role-attr-desc>
   |<role-attr-list> <role-attr-desc>

<simple-attr-desc> ::=
    <id>
    property : <property-values>
    default : <term>
    constraint : <pattern>
    type : <svs-name>

<role-attr-desc> ::=
    <id>
    property : <property-values>
    default : <number>
    constraint : <pattern>
    type : <data-class-name>

<term> ::=
    atom
   |list

<number> ::=
    integer

<property-values> ::=
    <property>
   |<property-values> , <property>

<property> ::=
    unique | optional | multivalued | onto

<class-attributes> ::=
    [<local-constraint>] [<general-constraint>] <operations>

<local-constraint> ::=
    entity local constraints : <expression>

<general-constraint> ::=
    general constraints : <expression>
```

```
<expression> ::=
    LISP expression

<operations> ::=
    predefined operations : <operation-list>

<operation-list> ::=
    <kbml-operation>
   |<operation-list> , <kbml-operation>

<kbml-operation> ::=
    $KB-CREATE
   |$KB-DELETE
   |$KB-CONNECT
   |$KB-DISCONNECT
   |$KB-RETRIEVE
   |$KB-GET
   |$KB-FETCH
   |$KB-REPLACE
   |$KB-ADD-ATTR
   |$KB-DEL-ATTR
   |$KB-REWIND
   |$KB-READ
   |$KB-PRINT
```

Appendix III

The BNF of the Knowledge Base Manipulation Language

```
<kbml> ::=
    ($KB-LOAD <class-name>)
   |($KB-UNLOAD <class-name>)
   |($KB-CREATE <class-name> [<attr-val-pairs>])
   |($KB-DELETE <entity-no>)
   |($KB-CONNECT <entity-no> <class-name>
                      [<attr-val-pairs>])
   |($KB-DISCONNECT <entity-no> <class-name>)
   |($KB-RETRIEVE <class-name> [<criteria>])
   |($KB-GET <entity-no> [<attr-list>])
   |($KB-FETCH <class-name> [<attr-list>] [<criteria>])
   |($KB-REPLACE <entity-no> <attr-val-pairs>)
   |($KB-ADD-ATTR <entity-no> <attr-name> <attr-value>)
   |($KB-DEL-ATTR <entity-no> <attr-name> <attr-value>)
   |($KB-BELONGS-TO <type> <value>)
   |($KB-REWIND <class-name>)
   |($KB-READ <class-name>)
   |($KB-PRINT <class-name>)

<class-name> ::=
    identifier

<attr-val-pairs> ::=
    <attr-val-pair>
   |<attr-val-pairs> <attr-val-pair>

<attr-val-pair> ::=
    (<attr-name> <s-expressions>)

<attr-name> ::=
    identifier

<attr-value> ::=
    <s-expression>

<s-expressions> ::=
    <s-expression>
   |<s-expressions> <s-expression>

<s-expression> ::=
    atom
   |list

<entity-no> ::=
```

```
    positive integer

<attr-list> ::=
    (<attr-names>)

<attr-names> ::=
    <attr-name>
   |<attr-names> <attr-name>

<criteria> ::=
    (<criterium-list>)

<criterium-list> ::=
    <criterium>
   |<criterium-list> <criterium>

<criterium> ::=
    (<attr-name> <expressions>)

<expressions> ::=
    <expression>
   |<expressions> <expression>

<expressions> ::=
    <s-expression>
   |<pml>  (see Appendix I)
```

References

[AHU 83]
Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. Data Structures and Algorithms. Massachusetts: Addison-Wesley Publishing Company, 1983.

[Brod84]
Brodie, M. L., et al, ed. On Conceptual Modelling. New York: Springer-Verlag, 1984.

[EiRa85]
Eick, C. F. and Thomas Raupp. Decentralized database design using multityped functional and existence dependencies, 1985.

[Eick86]
Eick, Christoph F. Data Management for LISP-Knowledge Bases, Submitted for publication, 1986.

[EnNi77]
Engelmore, R.S. and H.P. Nii. A Knowledge-based System for the Interpretation of Protein X-ray Crystallographic Data, Heuristic Programming Project Memo HPP-77-2, January, 1977.

[GoHa83]
Goos, G. and Hartmanis, J., ed. The Programming Language Ada Reference Manual. New York: Springer-Verlag, 1983.

[Grie82]
Griethuysen J.J. van, ed. CONCEPTS and TERMINOLOGY for the CONCEPTUAL SCHEMA and the INFORMATION BASE, 1982

[Haye74]
Hayes, P. J. Some Problems and Non-Problemx in Representation Theory, Proc. AISB Summer Conference, Essex Univ., Essex, Great Britian, 1974.

[LeEr77]
Lesser, V.R. and L.D. Erman. A retrospective view of the HEARSAY-II architecture, Proc. 5th IJCAI, 1977, pp. 790-800

[Mehl84]
Mehlhoun, Kurt. Data Structures and Algorithms 1:

Sorting and Searching. New York: Springer-Verlag, 1984.

[Nego85]
Negoita, C. V. Expert Systems and Fuzzy Systems. Massachusetts: The Benjamin Cummings Publishing Company, Inc., 1985.

[Schm77]
Schmidt, J.W. Some High Level Language Constructs for Data of Type Relation, ACM Transactions on Database Systems, Vol. 2, No. 3, September 1977.

[StWe85]
Stubbs, Daniel F., and Neil W. Webre. Data Structure with Abstract Data Types and Pascal. California: Brooks/Cole Publishing Company, 1985.

[Symb84]
Symbolics Inc., Symbolics 3600 Technical Summary. Massachusetts, 1984.

[Wile84]
Wilensky, Robert. LISPcraft. New York: W.W. Norton & Company, 1984

[Wins81]
Winston, Patrick, and Berthold Klaus Paul Horn. LISP. New York: Addison-Wesley Pulbishing Company, 1981.

[Youn83]
Young, S. J. An Introduction to Ada. New York: Ellis Horwood Limited, 1983.