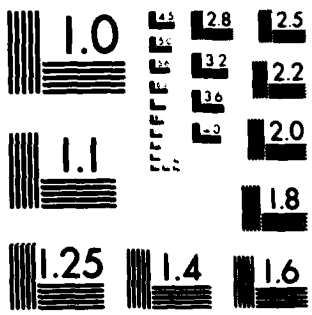
AD-A136 522 FILE SEARCHING PROBLEMS IN LOGIC PROGRAMMING SYSTEMS (U) FLORIDA STATE UNIV TALLAHASSEE DEPT OF MATHEMATICS AND COMPUTER SCIENCE C M EASTMAN FEB 83 1/1

UNCLASSIFIED AFOSR-TR-83-1252 AFOSR-81-0110 F/G 9/2 NL

END
DATE
FILMED
1-84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

ADA136522

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 83-1252
6a. NAME OF PERFORMING ORGANIZATION: Florida State University; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code): Department of Mathematics and Computer Science, Tallahassee FL 32306	7b. ADDRESS (City, State and ZIP Code): Directorate of Mathematical and Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: AFOSR-81-0110
8c. ADDRESS (City, State and ZIP Code): Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2304	A2	

11. TITLE (Include Security Classification): SEE REMARKS on Reverse

12. PERSONAL AUTHOR(S): Caroline M. Eastman

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM 1/3/81 TO 28/2/82	FEB 83	35

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

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

During this period, the investigator intended to investigate alternative approaches for improving searching performance in logic programming systems to a level that would be acceptable in a production system by conducting experiments in the LOGLISP system. Due to incompatibilities between the DEC 10 source computer and the CDC CYBER 760 running under NOS, which was available at Florida State University, as well as the differences between the UCI LISP on the DEC 10 and the ALISP available on the CYBER, it was impossible to bring the LOGLISP system to fully operational status and perform the experiments.

DTIC FILE COPY

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☒ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Robert N. Buchal	22b. TELEPHONE NUMBER (Include Area Code): (202) 767-4939; 22c. OFFICE SYMBOL: NM

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

ITEM #11, TITLE: FILE SEARCHING PROBLEMS IN LOGIC PROGRAMMING SYSTEMS

AFOSR-TR- 83-1252

FILE SEARCHING PROBLEMS IN LOGIC PROGRAMMING SYSTEMS

Caroline M. Eastman

February 1983

A final report on work performed under grant AFOSR-81-0110.

84 01 04 103

TABLE OF CONTENTS

LOGIC PROGRAMMING

1. Introduction

The area of knowledge representation has received active research interest recently as more powerful knowledge-based systems have been developed. Such systems show potential in several application areas, including database management systems, decision support systems, and automatic programming systems. A variety of techniques for knowledge representation have been explored; one of the more promising is the use of resolution logic.

Among the many problems which must be resolved before such approaches can be used in production systems is that of search efficiency. Current systems either handle a variety of problem structures at the cost of relatively unconstrained search or constrain search at the cost of rigidly defined problem structures. An additional search problem results from the need to expand the current small systems, which are primarily in-core systems, to much larger size.

2. Prolog

Prolog is a programming language based upon the use of resolution logic which provides a high level nonprocedural mechanism for writing programs and representing data. The Prolog language is described in Warren et al (1977), and the underlying logical theory is described in Kowalski (1974) and Robinson (1965). Kowalski (1979) provides an extensive introduction to logic programming with emphasis on Prolog.

The basic construct used in Prolog is the clause, which consists of a series of terms. An example of such a clause is

```
grandparent(x,y) :- parent(x,z), parent (z,y)
```

The first term is the head; the rest make up the body. If all of the terms in the body are true, then the head is true. A procedure is a set of clauses. A clause with an empty body is referred to as a unit clause. Suppose that the following unit clauses are added to the previous clause:

```
parent (Jim, Jane)

parent (Jane, Jennifer)

parent (Jane, John)

parent (Joe, John)

parent (Jim, Janice)
```

parent (Josie, Jane)

Then posing the goal grandparent (u, Jennifer) will find all of Jennifer's grandparents. Posing the goal grandparent (Jim, v) will find Jim's grandchildren. Posing the goal grandparent (u, v) will find all of the pairs of grandparents and grandchildren known to the system.

Consider the first case given, that of finding Jennifer's grandparents. The system attempts to show that the head is true by showing that both of the clauses parent (u, y) and parent (y, Jennifer) are true. (Here x is replaced by u, and z is replaced by Jennifer.) It can do this by substituting y = Jane and u = Jim or by substituting y = Jane and u = Josie. The process of finding appropriate substitutions for the variables is referred to as unification. The process of finding an appropriate unification means traversing a search space with many possible choices of clauses and variable assignments and includes the possibility of backtracking if a particular path does not work out. It is also possible that no appropriate unification will be found.

The example given here is quite simple. More elaborate examples can be found in the references previously given. Prolog can be used for a variety of applications ranging from intelligent databases to automatic programming systems.

3. Lisp

Lisp is a list oriented language based upon the lambda calculus. Two of the many expository descriptions are given in Greenberg (1978) and Siklossky (1976). A brief sumary of Lisp development is given in McCarthy (1978). Its predominant use is in artificial intelligence work.

In Lisp, both programs and data are represented as lists and are not explicitly distinguished. For example, a (very simple) Lisp program to evalate the square root of 3.3 + (4.1 x 5.2) could be written as

```
(SQRT (FPLUS 3.3 (FTIMES 4.1 5.2)))
```

This is a two item list; the second item is itself a list. When this function is evaluated, the multiplication in the inner sublist is evaluated first. Then the 3.3 is added. Finally the square root is taken.

Lisp contains a variety of functions and special constructs, including those for taking aprt and putting together lists, testing conditions, and manipulating numbers and strings.

4. Loglisp

Prolog and Lisp are not equally easy to use over a full range of applications. For example, the grandparent example used in the discussion of Prolog would be much harder to write in Lisp since a function which explicitly takes apart lists representing parent information would need to be written. On the other hand, the simple Lisp calculation given would be much harder to write in Prolog since real arithmetic and square roots are much harder to handle in a logic context.

Loglisp is a system which combines the advantages of both Prolog and Lisp. It has been developed under Air Force sponsorship (RADC) by Robinson and Sibert (1981) at Syracuse University as an extension to UCI Lisp for the DEC-10. In this language, Lisp is extended to allow the use of logic programming. The syntax and techniques are not quite the same as in the Prolog system, but the same basic capabilities are provided. The Lisp features and the logic features may be intermixed, or only one set of features may be used. So both pattern matching (as in Prolog) and function evaluation (as in Lisp) may be easily done in the context of an integrated language.

LOGLISP PORTABILITY

1. Loglisp Implementation

Loglisp was originally implemented at Syracuse University; a description of this system is given in Robinson and Sibert (1981). It was implemented as an extension to UCI Lisp, a dialect of Lisp 1.6, on the DEC-10 (Quam and Diffie, undated; Bobrow, Burton, and Lewis, undated). It has also been converted to run under INTERLISP. (Schrag, 1982).

2. The ALISP Dialect of Lisp

ALISP is a Lisp dialect based on Lisp 1.5 and described in The ALISP User's Manual (Univ. Mass., undated). It was developed at the University of Massachusetts at Amherst on Control Data mainframes. The first version ran on a CDC 3600/3800 under the UMASS timesharing system. The current version runs on CDC Cybers under the NOS operating system. In addition to basic Lisp features, it provides a compiler, a limited programming environment (editor, file system, and pretty-printer), and applications packages (relational database system and graphics routines).

Since the manual proved very difficult to use without an index, one was constructed. It is given in Appendix A. (The manual refers to its index, but it was not present and could not be located.) Since the feasability of conversion from UCI Lisp to ALISP was being examined, a comparison list of functions was constructed. It is given in Appendix B. This list shows functions present in each of the languages and includes comments on function similarities and differences. It can be seen from this list that there are substantial differences between the two languages.

3. Lisp Dialects

There are many dialects of Lisp in existence. There have been some efforts at standardization within the Lisp community, but these have not met with overwhelming success (e.g. Marti et al, 1979; Steele et al, 1982). While the basic core of the language is the same from dialect to dialect, the additional features provided as part of the system and the manner in which the system is implemented can differ widely. Such diversity is not surprising and perhaps healthy in an experimental language, but it has hampered the use of Lisp in more production-oriented environments.

Samet (1981) describes a conversion system to translate programs written in LISP 1.6 to INTERLISP which was motivated by the need to convert a program used in a compiler testing system. The conversion system was designed to run under either version of

LISP. It depends primarily on pattern substitution techniques and function redefinitions to convert LISP 1.6 features to INTERLISP, including most functions, I/O functions, the escape character, strings, names, and numbers. However, LEXPRs, macros, arrays, and a few functions were excluded from the conversion system.

Samet classified some problems as irreconcilable; these included mainly problems with differences in data type definitions. A working definition of an irreconcilable problem in this context is one which could not be handled by a straightforward transformation or which could not be handled without run-time support. The conversions that were performed were divided into those based on the external form of the program and those based on the semantics of basic constructs.

4. Software Conversion Studies

Conversion of software developed in one environment to another environment is an important activity (U.S. GAO, 1977, 1980). Despite the extensive resources devoted to conversion efforts, little formal attention has been paid to it. Most of our knowledge about software conversion is institutional in nature, based upon extensive experience.

Gilb (1977) defines software portability as "the ease of conversion from one environment to another; the relative conversion cost for a given conversion method or algorithm." He measures portability as 1-(ET/ER), where ET is the cost to transfer the system to a different environment, and ER is the original cost of developing the software.

This measure of portability depends not only on the system, but also on the two environments. The compatablity between the two environments can be measured as the average portability of systems converted from one environment to the other. Gilb's measures provide a measure of portability and compatability relative to a population of tasks, but they do not directly address the question of predicting the effort involved without guidance from previous experience with those systems.

Boehm (1981) tackles this more difficult question of estimating the resources required for conversion. His estimate of conversion costs is based on calculating a value for EDSI, equivalent delivered source instructions. This is estimated as

$$EDSI = (ADSI) \times (AAF/100).$$

Here ADSI is the actual delivered source instructions in the code to be converted, and AAF is an adaptation adjustment factor calculated as

$$AAF = (0.40 \times DM) + (0.30 \times CM) + (0.30 \times IM).$$

Here DM is an estimate of the percent of the design modified, CM is an estimate of the percent of lines of code wich must be modified, and IM is an estimate of the percent of the original integration and testing which must be performed on the converted software.

The total conversion effort in man-months can then be estimated as

MM = 2.4 x (EDSI)**1.05.

Boehm includes an extensive discussion of cost driver factors wncih can be usd to take into acccount such factors as system complexity, reliability requirements, programming language, and staff experience.

Obviously these estimates, especially for DM and IM, must be in large part subjective. However, this approach provides a structured framework for the problem of estimating software conversion effort whose worth is supported by extensive experience.

5. Specific Problems

A number of incompatability problems betweeen UCI Lisp and ALISP were encountered. These were categorized as environment incompatabilities, feature incompatabilities, syntactic immcompatabilities and fundamental incompatabilities.

The environmental incompatabilities included

character set

The character sets used in the two systems are different both in size and in encoding. This created some delay in even reading a Loglisp tape. Furthermore, some characters used for special purposes in Loglisp are referred to by their encoded value (CHRVAL).

editor

Both systems included an editor as part of the environment. These allow both structure editing and pattern matching. However, the commands used are different. The editor is used by Loglisp in order to handle editing of knowledge bases. Formatters are provided in both systems, but different function names are used.

file system

Both dialects of Lisp provide functions to allow access to the systems file system in order to facilitate file handling with the Lisp system. The capabilities provided are similar, but the underlying file systems are not.

Feature incompatabilites included

<u>lack of function correspondence</u>

There are many functions present in UCI Lisp which have no corresponding functions in ALSIP. For the most part these can be handled in a straightforward manner simply by writing new function definitions. Appendix B contains a list of the functions present in both Lisp dialects.

<u>macros</u>

UCI Lisp provides a macro capability; ALISP does not. Features implemented using macros thus need to be rewritten.

Syntactic incompatabilites included

<u>inconsistent function names</u>

In many cases different names were used in the two dialects for the same function. Examples include ABSVAL (ALISP) vs. ABS (UCI Lisp) and DIFF (ALISP) vs. DIFFERENCE (UCI Lisp). These problems are also quite straightforward to handle by renaming.

<u>inconsistent function syntax</u>

In a few cases the syntax used for functions was not consistent. For example, the parameter order for MAPC is different. Although handling these situations is straightforward, they are more subtle since they appear correct.

Fundamental incompatabilites included

<u>handling of function definitions</u>

In ALISP function definitions are stored in the value cell of the appropriate literal atoms. In UCI Lisp function definitions are stored on the property list. Thus function definitions are easier to change on the fly in UCI Lisp. Since this is done in Loglisp in order to switch between Lisp and Logic, substantial conversion problems are presented.

It should be noted that, with very few exceptions, these problems are inherent in the dialect differences and are not due to the design and implementation of Loglisp. It would be extremently difficult to implement a system with complex functionality without making full use of the features available.

6. Language Similarity

The similarity between ALISP and UCI Lisp can be examined by comparing the functions available in the two languages. A simple measure of such similarity between two programming languages is given by

$$SIM = OV / (N1 + N2 - OV)$$

where N1 is the number of function names used in one language, N2 is the number of function names used in the other language, and OV (overlap) is the number of function names used in both languages. This measure ranges between 0 (least similar) and 1 (most similar).

There are a number of factors which are not taken into account by this measure. Function names in Lisp and other programming languages are not used with equal frequency, and the most frequent ones should perhaps be given higher weights. Also, no distinction is made between a function name used for a function present in one language but not the other and a function name used for a function which is present in both languages but called by different names.

There are 303 function names for UCI Lisp and 227 for ALISP; these are given in Appendix B. There are 80 overlaps, including most of the "core" Lisp functions. The similarity is 0.18. It is interesting to note that the similarity computed by the same measure between COBOL and Ada is 0.32 (Eastman, 1982). So the keyword similarity between two dialects of the same language can actually be less than that between two distinct languages.

7. Estimate of Conversion Effort

The formulas given in Boehm were used to estimate the conversion effort required for a full conversion of Loglisp to the ALISP system. The Loglisp system contains approximately 2,000 lines of code, as formatted by the pretty-printer. The factors DM and IM are estimated at 30% to allow for the change in the system environment and function definition mechanism as well as the more straightforward changes. CM is estimated at 50%. With these figures, EDSI is estimated at 720 and conversion effort in man-months at 1.7. The tape conversion required about 0.25 MM (Franson and Haslup, 1981), and the conversion of a minimal core system required about 0.5 MM.

Of course, these figures provide only a very rough estimate of conversion effort. They were developed based on experience with systems written in other languages, and it is not at all clear how well they apply to Lisp. Since programming in Lisp is more complicated than programming in COBOL or FORTRAN, it is likely that these formulas will underestimate the effort required. Also, the concept of "line of code" is not as well defined for Lisp as it is for many other languages since programs are not divided into statements in the same way as for many languages. The approximation used here was to simply use the lines of text provided by the prettyprinter; however, the line breaks could have been done in many different ways. It would be highly desirable to have a working definition of source instruction for Lisp that could be used in such estimations and to have data validating their use in a Lisp environment.

FILE ORGANIZATIONS FOR LOGIC PROGRAMMING

1. Introduction

The clauses in a logic program can be divided into the two categories of rules and facts. A fact is defined here as a clause with no body and no variables. A query is a clause containing only a body. All other clauses are rules. So a fact will be at the end of an inference path and will not lead to further steps. A rule contains either variables or inferences (or both) and can lead to a further unification step. This division follows that made by Klahr (1979) and corresponds closely to similar (but not necessarily identical) distinctions made by others (e.g. unit and non-unit clauses, ground and non-ground clauses, extensional data base and intensional data base, assertions and implications.)

Searching in logic based systems can thus be broken down into two distinguishable but related searching problems: rule selection and fact retrieval. Rule selection involves the choice of the next clause or goal to use in the inference process. Fact retrieval involves locating a particular clause. However, the distinction is not absolute. Located facts are used in the inference process, and selecting rules involves finding them as well. Fact retrieval is generally discussed within the context of the retrieval paradigm used in the database area; overviews are provided in Knuth (1977) and Wiederhold (1977). Rule selection falls within the heuristic search paradigm used in artificial intelligence. In most current work, these two searching problems are kept separate; this approach appears to be more efficient than intermixing them.

2. Database Applications

The relative importance of rule selection and fact retrieval depends in large part on the application. Some applications, such as database systems, have relatively few rules. Others, such as theorem proving systems, have relatively more rules. Since searching problems in unrestricted resolutions sytems are not yet well understood, it is reasonable to consider only a subset of such problems. One way to narrow the problem is to consider application areas of interest to see what the implications of their specific characteristics are for search strategies.

Database applications are considered here since use of artificial intelligence techniques can provide databases with more flexibility than is possible in current commercial systems. A possible definition of an intelligent data base is that it is a database from which implicit information which is not explicitly stored can be retrieved. One well-studied class of such databases allows the use of logical inference. Such inferential databases are discussed in Gallaire and Minker (1978), Klahr

(1979), and Minker (1978). (It is possible to have other types of inference, e.g. statistical.)

Some of the characteristics to be expected of an inferential database based on logic principles are

a. Low ratio of rules to facts

b. Low level of most proofs

c. Possible need for best match and range searches

d. Rapid response for interactive use

e. Possible need for extensive updating

f. Potentially very large size

g. Varying degrees of data accuracy and timeliness

h. Not all relevant data included (open world assumption)

These characteristics are not necessarily found in other logic application aras. For example, none are likely to be true of theorem proving applications. These factors influence the acceptable searching strategies.

3. Limitations of Hashing

C -rent artificial intelligence systems depend heavily on hashing. When it is feasible, hashing is extremely fast and is often the method of choice. This reliance upon hashing has allowed researchers to concentrate on the problem structure and heuristic search without getting bogged down in the details of fact retrieval. And this approach has not created any problems in the relatively small in-core systems predominant in artificial intelligence work. However, hashing has some limitations which may cause problems in such systems; thse involve update problems, best match and range searching, and locality of reference.

Hash indices degenerate under update to a greater extent than many other index systems (e.g., B-trees). This requires more garbage collection and possible rehashing. While the pointer chasing involved in Lisp garbage collection does not present a problem in in-core systems, it may in systems that involve storage over several levels of memory hierarchy.

A second problem is that, unless the hash function used is order preserving, it is very hard to handle situations involving best match or range searches. The respond to such a request, the system is reduced to hashing on all possible values that might fall within the desired range or to a sequential search. Index techniques which preserve order information are much better able to handle such questions.

Management of a memory hierarchy is far more effective when there is a high degree of locality of reference. This allows the I/O overhead to be spread over a greater number of accesses. large systems will presumably depend on the use of a memory hierarchy and will need to use storage techniques which preserve locality of reference.

It would be desirable to exploit the advantages of hashing even in a much larger system. However, mechanisms which preserve order and locality will be required.

4. Choice of File Organization

Almost all of the work on selection of file organizations has focused on situations in which the workload to be handled is both precisely known and stable. Under those conditions, there are many algorithms know which can be used to determine optimal file organizations and/or parameters. However, these approaches are not necessarily as useful when the workload is neither precisely known or stable, as is to be expected in an intelligent database.

In such a context only partial information will be available about the classes and frequencies of queries which will be posed to the system, the fields which will be used in queries, and the frequencies with which those fields will be used. The relative importance of update and retrieval is also unlikely to be known precisely.

Consider a ground relation which has five possible fields which could be used for indexing. If the frequencies with which the different combinations of fields will be used in retrieval and update are known, it is possible to determine a reasonable solution to the problem of determining which of the fields should be indexed (e.g. Anderson and Berra, 1977).

However, if little is known about the workload that the system will handle, such algorithms are not directly applicable. They must be supplemented by other approaches, which might include worklad sampling, risk analysis, statistical analysis, possibility theory, and expert systems. The system can maintain statistics on requests posed over time in order to estimate the workload at a given time. This information can be used to select an optimal or near optimal configuration based on this information. However, this may not be adequate if the workload is highly variable. The use of logic procedures to analyze useage data and determine appropriate storage organizations is consistent with the goal of an integrated logic system.

It will be difficult to extend the size of current intelliient systems by one or more orders of magnitude without the use of different hardware architectures. The are two main directions in this architectural work: the use of multiple processors and the use of content addressable (or associative) storage. Multiple processors, which could be in a distributed system, could be used to explore, in parallel, alternate paths through the search space. Associatie storage can be used for fast retrieval. (Schrag 1981) describes a possible [illegible] architecture for handling logic database applications. Even if a computer architecture well adapted to these types of applications is used, it will still need to be used effectively. So storage organizations will contiue to be a problem even though the problem parameters will change.

A LOGIC DESIGN FOR A DOCUMENT RETRIEVAL DATABASE

1. Information Retrieval Systems

The term information retrieval is widely used in both a broad sense and a narrow sense. In its narrower meaning, an information retrieval system is a reference retrieval system. It is this use which will be considered here. Such systems provide lists of references to documents in response to user queries based on topics, authors, dates of publication, and similar types of information. Information retrieval systems differ from database management systems for formatted data in their emphasis on textual data.

The index terms used to retrieve the documents may be chosen from a carefully controlled thesaurus of allowed terms. Or the important words from the title, the abstract, or the entire text may be used in a less controlled situation. Normalization to a standard form is generally allowed in order to eliminate variation caused by plurals, verb endings, and other suffixes. In current commercial systems, boolean combinations of index terms are generally used in queries. More elaborate matching functions have been used in experimental systems, and research indicates that they might be more effective than conventional boolean matching.

Information retrieval systems are in comon use today. They range from extensive comercial services, such as Lockheed's DIALOG system, which provide access to dozens of different databases (collections of documents) to small systems intended for personal use to experimental systems. An introduction to information retrieval systems may be found in Salton and McGill (1983).

Evaluation of information retrieval systems is based on perormance effectiveness as well as efficiency. The response to a query for documents on a particular topic will generally include some items which are not relevant to the query and fail to include some items which are relevant. The extent of such failures is measured by recall and precision. Recall is defined as the fraction of the relevant documents which are retrieved. Precision is defined as the fraction of the retrieved documents which actually are relevant.

2. Databases Implemented in Logic

Extensive work has been performed on the application of logic to databases. The volume edited by Gallaire and Minker (1978) contains many primarily theoretical papers on the subject. Dahl (1982) describes a database application with a Spanish language interface which was implemented in Prolog. The example application is a small employee database.

To describe a database, Dahl ues domain definitions, which also include hierarchical relationships, relation declarations, to serve as relation templates, and relation definitions, which include the actual data. Coelho (1979) describes an adaptation of this approach to Portuguese.

3. Basic Features of an IR System

Documents in an information retrieval system need to be described by their contents and by the appropriate bibliographic information. A possible set of relation declarations for this information is

title (document title)

author (document author order)

date (document publication-date)

publisher (document publisher)

topic (document concept)

The first four relations (title, author, date, and publisher) provide the basic bibliographic information for the document. Here it is assumed that, for a given document, there is only one title, one publication date, and one publisher. There may however, be more that one author. The order is preserved for an author so that it is possible to tell the order in wnch the authors were listed.

The domain for document is assumed to be a set of unique document identifiers. It is not possible to reliably identify a document by author, by title, by publisher, or even by date; so a unique identifier is required. (The need for unique identifier for the various entities about which information is stored has been glossed over in work on logic databases.)

A document may contain information about many topics, and a topic may be discussed in many documemts. It is possible to obtain all the topics contained in a document or all the documents discussing a topic by posing the symmetric queries:

<- topic (document x)

<- topic (x concept)

where either a specific document or a specific concept is indicated.

A bibliographic description of a document may be defined using

```
description (u v w x) <- title (u v),
                         publisher (u w),
                         date (u x)

<- author(document x y)
<- description (document v w x)
```

Usually a combination of topics is specified in a query rather than a single topic. Boolean queries involving the operators AND and OR could be readily handled using the Loglisp boolean operators or other extralogical features to avoid repetition in query specification.

4. Other Features

Retrieval performance can be enhanced by the use of synonym information. Even if one term is used in a query, documents classified with synonyms of that term can be retrieved. This feature could be handled by the following ground relation

```
synonym (word1 word2)
```

and the following rules

```
synonym (x y) <- synonym (y x)

synonym (x y) <- synonym (x z), synonym (z y)

topic (document x) <- topic (document y), synonym (x y)
```

A more elaborate thesaurus could be implemented to handle hierarchical relationships.

Concept weights are frequently used in order to indicate the relative importance of topics discussed in document. Such concept weights could be handled by using an expanded ground relation

```
topic (document concept weight)
```

A query which request documents which are about a specific topic can then specify a cutoff weight

```
<- topic (document concept x), greater (x cutoff)
```

Since, in most systems, concepts with zero weights are unlikely to be explicitly entered, testing the weight to see if it is greater than zero should be unnecessary.

There is some evidence that retrieval performance can be enhanced by using matching functions other than simple boolean matchig. Suggestions include the use of similarity functions based on vector models of documents and queries and the use of fuzzy set theory. Such measures could be implemented in logic but might be more easily implemented in extralogical features such as those available in Loglisp.

The closed world assumption that all relevant information is present in the knowledge base is not viable in information retrieval systems. It is generally accepted that a document may contain information about a concept even though that concept has not been used to index it. Thus failure can not be interpreted as negation.

REFERENCES

Henry D. Anderson and P. Bruce Berra, "Minimum cost selection of secondary indices for formatted files", ACM Transactions on Database Systems, Vol 2, No 1, March 1977, pp 68-90.

Robert J. Bobrow, Richard R. Burton, and Daryle Lewis, UCI LISP Manual, undated.

Barry W. Boehm, Software Engineering Economics, Prentice-Hall, I E 'ewood Cliffs, New Jersey, 1981.

Helder Coehlo, "A program conversing in Portuguese providing a library service", PhD thesis, University of Edinburgh, December 1979.

Veronica Dahl, "On database systems development through logic", ACM Transactions on Database Systems, Vol 7, No 1, March 1982, pp 102-123.

C. M. Eastman, "A lexical analysis of keywords in high level programming languages", submitted for publication.

John Franson and Lee Haslup, "Implementing Loglisp at FSU", Florida State University, 1981.

Herve Gallaire and Jack Minker (editors), Logic and Data Bases, Plenum Press, New York, NY, 1978.

Tom Gilb, Software Metrics, Winthrop Publishers, Cambridge, Massachusetts, 1977.

P. Klahr, "Conditional answers in question-answering systems", Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Vol 1, pp 481-483, 1979.

D. E. Knuth, The Art of Computer Programming, Volume III: S rching and Sorting, Addison-Wesley Publishig Company, Reading, Massacusetts, 1973.

Robert Kowalski, Logic for Problem Solving, Elsevier North Holland, Inc., New York, NY, 1979.

Robert Kowalski, "Predicate logic as programming language", Proc IFIP 74, North Holland Publishing Co., Amsterdam, pp 569-574, 1974.

J. McCarthy, "History of LISP", SIGPLAN Notices, Vol 1, No 8, pp 217-224, 1978.

J. Marti, A. C. Hearn, M. L. Griss, and C. Griss, "Standard LISP report", SIGPLAN Notices, Vol 14, No 10, October 1979, pp 48-68.

J. Minker, "Search strategy and selection function for an inferential relational system", ACM Transactions on Database Systems, Vol 3, No 1, pp 1-31, 1978.

Lynn H. Quan and Whitfield Diffie, Stanford LISP 1.6 Manual, Stanford University, undated.

J. A. Robinson, "A machine oriented logic based on the resolution principle", Journal of the ACM, Vol 12, pp 23-41, 1965.

J. A. Robinson and E. E. Sibert, "Logic programming in LISP", RADC-TR-80-379, Vol 1, Rome Air Development Center, Griffiss Air Force Base, New York, 1981.

Gerard Salton and Michael McGill, Introduction to Modern Information Retrieval, McGraw-Hill Book Company, New York, NY, 1983.

Hanan Samet, "Experience with software conversion", Software--Practice and Experience, Vol 11 105-1069, 1981.

Robert C. Schrag, "Logical data base machines", Syracuse University, 1981.

Robert C. Schrag, personal communication, 1982.

Guy L. Steele, Jr. et al, "An overview of Common Lisp", ACM Symposium on Lisp and Functional Programming, 1982.

U. S. General Accounting Office, "Millions in savings possible in converting programs from one computer to another", FGMSD-77-34, 1977 (referenced in U. S. GAO, 1980)

U. S. General Accounting Office, "Wider use of better computer software technology can improve management control and reduce costs", FGMSD-80-8, 1980.

University of Massachusetts, ALISP Reference Manual, University of Massachusetts at Amherst, undated.

D. H. D. Warren, L. M Pereira, and F. Pereira, "PROLOG--the language and its implementation compared with LISP", SIGPLAN Notices, Vol 1, No 8, pp 109-115, 1977.

G. Wiederhold, Database Design, McGraw-Hill Book Company, New York, NY, 1977.

G. A. Wilson and J. Minker, 'Resolution, refinements and search strategies: a comparative study", IEEE Transactions on Computers, Vol C-25, No 8, pp 182-801, 1976.

APPENDIX A

INDEX TO THE ALISP REFERENCE MANUAL

ALISP INDEX

ALISP INDEX

ALISP INDEX

ALISP INDEX

ALISP INDEX

APPENDIX B

A COMPARISON BETWEEN ALISP AND UCI LISP

UCI LISP	ALISP	COMMENTS
abs		absval
	absval	abs
acos (u)		
addl		
	addel	
	addgt	
	addlt	
alist		
and	and	
and= (u)		
append	append	
apply	apply	
	apply*	
apply= (u)		
arg		
	argn	
array	array	
	arrayl	
	arrayp	
	arrtype	
ascii	ascii	
asin		
assoc(u)		
assoc= (u)		
atan (u)		
	atlength	
	atmhasn	
atom		
	backprn	
	backtrk	
bakgag		
base		
bignum		
	bnum	
boole		logand, logor, logxor
bpend		
bporg		
break	break	
breakl (u)		
breakd (u)		
breakexp (u)		
breakin (u)		
breakmacros (u)		
brokenfns (u)		
c....r	c....r	
car	car	
	cars	
cdr	cdr	
	cdrs	
chrct		
chrval (u)		
	close	
clrbfi (u)		

UCI LISP	ALISP	COMMENTS
	clrbit	
	comment	
	compile	
	conc	
	concons	
cond	cond	
cons	cons	
consp (u)		
copy (u)	copy	
	copyfile	
cos (u)	cos	
cosd (u)		
cosh (u)		
csym		
	cshift	
	dcopy	
ddt (u)		
ddtin		
ddtout		
de	de	
	decfile	
defprop	defprop	
	deletel	
deposit		
df	df	
	diff	difference
difference		diff
	dims	
	dispose	
divide	divide	
dm		
	do	
	docons	
dremove (u)		
dreverse (u)		
drm (u)		
dskin (u?)		
dskout (u)		
dsm (u)		
dsubst (u)		
	echo	
ed		
edfun		
	edit	
editcomsl (u)		
editdefault (u)		
edite (u)		
editf (u)		
editfindp (u)		
editfns (u)		
editfpat (u)		
editl (u)		
editp (u)		

UCI LISP	ALISP	COMMENTS
editracetn (u)		
editv (u)		
editye (u)		
	efface	
	entry	
	eofmak	
	eofskip	
	eofstat	
	eoir	
	eoiw	
eq	eq	
	eqp	
	eqs	
equal	equal	
err	err	
	errprin	
error (u)		
errse (u)		
errset	errset	
	esnirt	
eval	eval	
	evalquote	
evalv (u)		
examine		
exarray		
excise		
	exit	
exp (u)	exp	
explode		
explodec		
expr		
fexpr		
filein		
filenames		
	filestat	
fix	fix	
	fixp	
fixla		
fixnum		
	flambda	
flatsize		
flatsizec (u)		
float (u)	float	
	floatp	
flonum		
force		
fndbrkpt (u)		
	fntype	
free (u)		
freelist (u)		
	frunno	
fsubr		
funarg		
function		

UCI LISP	ALISP	COMMENTS
functional		
	fuzz	
gc	gc	
gcd		
gcgag		
gctime		
	genchar	
gensym		
	get	
	getfun	
	getval	
go	go	
greaterp	greaterp	
	grind	
grindef		
grinl (u)		
grinprops (u)		
	gts	
	halfpri	
hghcor (u)		
hghend (u)		
hghorg (u)		
	hprnum	
	hw	
ibase		inbase
identifier		
	if	
	illegal	
	inbase	ibase
inc		
	initfile	
initfn		
initpromp (u)		
input	input	
	intadd	
integer		
intern	intern	
	inunit	
label	label	
lambda	lambda	
lap (u)	lap	
laplst (u)		
last	last	
lastword (u)		
lastpos (u)		
lconc (u)		
ldiff (u)		
length	length	
lessp	lessp	
lexorder (u)		
lexpr		

UCI LISP	ALISP	COMMENTS
linelength		
lineread (u)		
list	list	
	listfile	
litatom (u)		
	litp	
	lnum	
load	load	
log (u)	log	
	logand	boole
	logical	
	logp	
	logor	boole
	logxor	boole
lsh		
lsubr		
lsubst (u)		
	lts	
macro		
maknam		
maknum		
map		
mapc	mapc	different parameter order
mapcan (u)		
mapcar	mapcar	
mapcon (u)	mapcon	
mapconc (u)	mapconc	
	mapl	
maplist	maplist	
max (u)	max	
maxlevel (u)		
memb (u)	memb	
member	member	
member= (u)		
memq		
memq= (u)		
min (u)	min	
minus	minus	
minusp	minusp	
modchar (u)		
nconc	nconc	
ncons	ncons	
neq (u)		
nextev (u)		
nil	nil	
nill (u)		
nocall (u)		
	normtab	
not		
nouuo		
nth (u)		
nthchar (u)		
null	null	

UCI LISP	ALISP	COMMENTS
number		
numberp	numberp	
numval		
oblist	oblist	
	oddp	
	open	
or	or	
or= (u)		
	outbase	
outc		
output	output	
	outputa	
	outunit	
outval (u)		
	pack	
	packl	
	pagefile	
	paramfi	
	paramgc	
	paramti	
patom (u)		
pgline		
	plist	
plus	plus	
	plusp	
pname		
	pprine	
	pprint	
prevev (u)		
prin1	prin1	
	prinarray	
	prinb	
	prinbeg	
princ		
	prinend	
	prinlen	
print	print	
printlev (u)		
prog	prog	
prog1 (u)		
prog2		
progn	progn	
prompt (u)	prompt	
	prop	
	purge	
	purgfile	
	put	
putprop		
putsym		
	qsetq	
quote		
quotient	quotient	

UCI LISP	ALISP	COMMENTS
random (u)		randy
	randy	random
read	read	
	readarray	
	readbeg	
readch	readch	
	readend	
	readent	
	readlen	
readlist		
	readnp	
	readnm	
readp (u)		
	readpk	
ree (u)		
remainder	remainder	
remob	remob	
remove (u)		
remprop	remprop	
	repeat	
reset		
retrrom (u)		
return	return	
reverse	reverse	
	rewind	
rplaca	rplaca	
rplacd	rplacd	
	runtime	
sassoc		uses eq
	save	
selectq (u)	selectq	
set	set	
setarg		
	setbit	
setchr (u)		
	setq	
sin (u)	sin	
sind (u)		
sinh (u)		
	slashes	
speak		
specbind		
special (u)	special	
specstr		
spdlrt (u)		
spdlpt (u)		
spdlrt (u)		
spredo (u)		
spreval (u)		
sprint		
sqrt (u)	sqrt	
	status	
stkcount (u)		
stkname (u)		

UCI LISP	ALISP	COMMENTS
stknth (u)		
stkprt (u)		
stksrch (u)		
store		
	strcars	
	strcdrs	
	strconc	
	strfind	
stringp (u)		
	strtest	
sub1	sub1	
sublis (u)		
subpair (u)		
subr		
subst		
	sys	
sysclr (u)		
	sysin	
	sysprin	
	sysout	
t		
tab (u)		
tailp (u)		
tan (u)		
tanh (u)		
tconc (u)		
	teread	
terpri	terpri	
time		
times	times	
	togbit	
trace	trace	
tracedfns (u)		
tracet		
	tracflg	
	tstbit	
	ttychar	
ttyecho (u)		
tyi		
tyo		
unbound (u)		
unbreak (u)		
undolst (u)		
unfind (u)		
	unitnos	
	unpack	
untrace (u)	untrace	
untyi (u)		
upfindlg (u)		
	valuep	
	wipe	

UCI LISP	ALISP	COMMENTS
	wipelist	
xcons		
zerop	zerop	
eof		
	*	
*amake		
*append		
*dif		
*eval		
*expand		
expandl		
*function		
*getsym		
*great		
*lcall		
*less		
*max (u)		
*min (u)		
*nopoint		
*plus		
*putsym		
*quo		
*rset		
*times		

DAT
FILM