A QUERY SYSTEM FOR INFORMATION RETRIEVAL IN A MAILBOX

by

SONG HEE KIM

B.A., Ewha University, Seoul, Korea, 1976

A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

Richard A. McBride

Major Professor

ACKNOWLEDGEMENTS

The author wishes to express her gratitude to Dr. Richard A. McBride, Assistant Professor of Computing and Information Sciences for his guidance and advice throughout this work. Thanks are also due to the members of the advisory committee, Dr. Elizabeth A. Unger, Professor of Computing and Information Sciences, and Dr. David A. Gustafson, Associate Professor of Computing and Information Sciences for their suggestions in preparation of this report.

The author's husband, Yong and her children Kihoon and Jane deserve special thanks. Their love and understanding and encouragement have become a great strength to get through this work. The author also wishes to express her deep thanks to her parents, Mr. Chang K. and Mrs. Sun H. Han, Mr. Dong K. and Mrs. Eun S. Kim, for their assistance which initiates and makes this study possible.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. BACKGROUND AND JUSTIFICATION

In recent years, there has been a substantial increase in the number of individuals and organizations using electronic mail facilities. Most electronic mail systems including those of the Unix system provide the user with facilities to create, send, receive, save and retrieve messages. The 4.3 BSD Unix mail system provides the user with several methods for accessing mail. Given the current usage of the Unix mail facility, it is evident that the improvement and enhancement of the Unix mail system is a matter of considerable importance.

At the simplest level, it would be desirable that a user should be able to retrieve messages via the following message items:

--- Date of receipt

--- Date of sending

--- Sender's identity

--- Recipient's identity

--- Subject of the message

--- Words or a phrase in a message body.

The QIRM (Query system for Information Retrieval in a Mailbox) system, which is designed by the author and described in and implemented in the present work, relies on searching for information in the headers or the bodies of messages which have been already saved in the system mailbox file. The QIRM system is capable of searching the complete text of all messages for words or phrases specified by the user. The retrieval functions of this system are flexible enough to permit the user to categorize the desired message(s). The requested information is retrieved according to the fields and conditions specified.

A query language provides a convenient scheme for retrieving information from databases. If we view the mailbox as a repository of stored messages (i.e., as a database file), we can employ queries expressed in some 'mail-query language' to extract information from the mailbox.

In the present application, Lisp has been chosen as the host language for implementing the mail query language. One major factor which motivated this decision is the symbol-manipulation capabilities provided by Lisp. A central role is played in QIRM's implementation by a semantic net data structure which determines the correspondence between symbols and the roles they play within a message. Lisp maintains all of the properties and values together in a property list associated with each atom. These property lists constitute a simple kind of database. For an example of the use of property lists, consider a database of information about the

messages in a mailbox. A property list associated with each message (atom) could be used to cache its property values with corresponding property names (e.g., sending date, receiving date, sender's id-name, recipient's id-name, status, subject, and message body). The global variable MAIL could be used to hold the list of all the messages in a mailbox. By employing the various facilities provided by Lisp, the data from a 'mailbox file' can be embodied in a database by using the concept of 'semantic nets'.

In order to enhance the functionality of the mail system, the messages which the system handles have to be interpreted at least partially. In this way, users can query the message system to find messages. The message system needs some guidance if it is to interpret messages correctly. Such guidance can be provided by superimposing some structure on the messages. The breakup of the header into fields and the interpretation of each field provide a suitable structure which can be used for locating messages. Since the facility allows the user to specify the partial contents of a message, an additional structure based on the message contents can be superimposed. This structure, which relies on the various information fields of messages, can be implemented by employing several concepts and techniques of Artificial Intelligence such as frames, production rules, inheritance [4], and semantic nets.

Even though QIRM is based on such primitive techniques of artificial intelligence as semantic nets, property lists and

matching, the prototype system could be used as the first stage in the design and development of an intelligent system for electronic mail handling. The approach which treats messages according to their contents encourages the integration of a message system with office functions performed on messages. In addition, this approach allows the integration of message and database facilities. Such versatility is one of the goals of office automation [5].

1.2. REPORT ORGANIZATION

The rest of this report is organized into three main chapters. Chapter 2 provides a review of the literature dealing with the concepts employed in the present implementation. A brief overview of electronic mailbox systems is provided along with definitions of relevant terms. Some notable aspects of query languages, semantic nets, and the language used to implement the project are also presented. At the end of that chapter, research work related to this report is summarized. Chapter 3 deals with the design of the QIRM system. Along with the presentation of the objectives of the system and the environment employed, there are descriptions of how the database is constructed, how the interface to the query system works, and how the tasks of data searching, extraction, and output generation are performed. Chapter 4 offers concluding remarks and suggests future extensions of this work.

There are two appendices. Appendix A is a manual

providing the syntax and user interface of this system. Appendix B is a listing of the source code of the implementation.

CHAPTER 2

REVIEW OF THE LITERATURE

2.1. ELECTRONIC MAILBOX SYSTEMS

This section consists of three parts. The first part defines the terminology related to the concept of an electronic mailbox system. In the second part of this section, the basic mailbox facilities are briefly described. Various approaches towards structuring mailbox facilities are presented in the last part. One of these approaches forms the basis for the implementation of the QIRM system.

2.1.1. Terminologies

There is some inconsistency in the use of terminology such as Electronic Mail, Mailbox, Electronic Message Systems. In fact, different people use each of these terms to mean different things. Consequently, it is not possible to provide clear cut, commonly accepted definitions for them. Another problem is that the technology is moving so rapidly that the concepts and terminology have to change quite often just to keep pace. Nevertheless, in order to place some of these terms in context, we shall define them by combining definitions in several references[1, 6, 10, 11, 20] based on an office automation approach.

Message refers to a single letter from a sender to be transferred via the system. It usually has two parts: an

envelope and contents. The envelope contains information needed by the system to get the message to the correct mailbox and typically contains the name and address of the mailbox to which the message is being sent. The contents can be separated into two parts: a header and a body. Header information contains predefined fields associated with the messaging process such as 'subject', 'time sent', 'to' and 'status'. A status-field contains such an information as 'Has the message been read or not?' or 'Is the message old or new?'. The system will request any essential envelope and header information before sending a message. The message body is free-format text. Some kind of word processing facility is usually provided for the sender to input and edit the body of the text at will (see Figure 1).

Electronic Mail is a collection of electronic text messages to be transferred uni-directionally, via a computer-assisted communication system, from an identified sending party to one or more identified receiving parties. It must be mentioned that the term is often used to refer to the electronic distribution of complete documents, composed of text, data, images and other forms of information.

Electronic mailbox systems or Electronic mail systems are used to describe computer-operated message systems which hold messages in mailboxes, thereby allowing the user to access and send messages at times and places which the user chooses.

Mailbox is a term sometimes used to refer to a store of messages, or to a program which provides users with access to

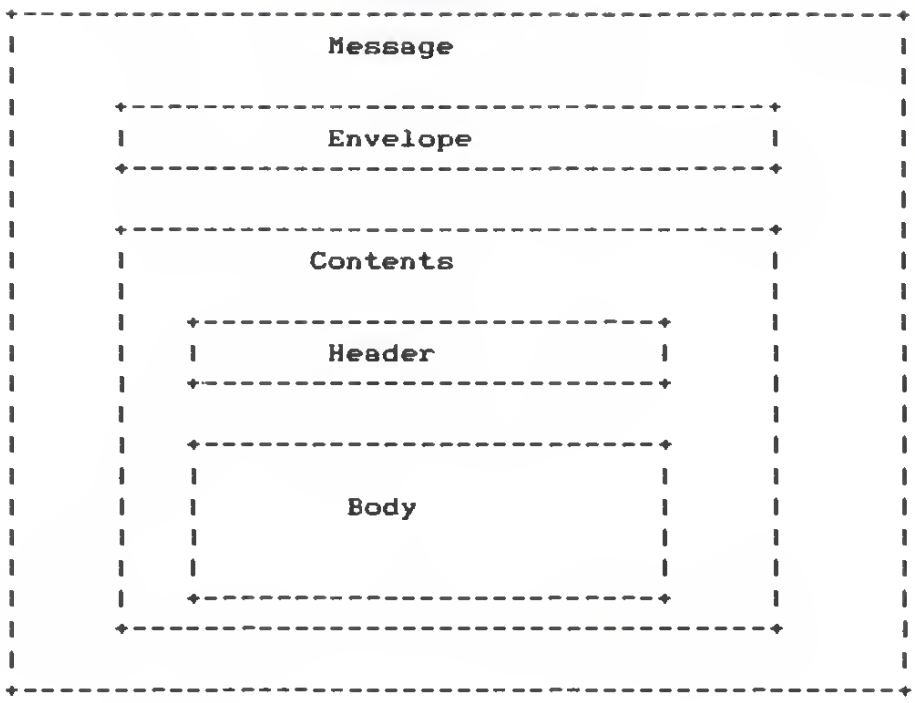

Figure 1. Message Structure

these messages. We shall use the term "mailbox" to refer to a file of incoming messages. In the Unix system, mailboxes are typically located in files corresponding to users' login names existing in the directory "/usr/spool/mail". Along with a mailbox, the system has the facility of stored, delayed communication. Nondelayed systems do not provide storage capacity. Therefore, they only support instantaneous communication. Conventional telex and telephone systems, and the direct terminal-to-terminal working on a multi-access computer system are non-delayed mail systems. Store-and-forward telex, telephone answering systems and electronic mail systems are examples of delayed systems.

Electronic Message Systems (EMS) refers to the entire range of electronic communication systems. EMS provides communications service to users based on the transmission of text, voice, image, or any combination of these three. Thus, Facsimile, Video Conferencing, PABX-based telephone systems, Voice mailbox systems, Telex, and Teletex are examples of EMS. However, the term EMS is sometimes used to specificially refer to Electronic Mailbox Systems.

2.1.2. Basic Mailbox Facilities

All Electronic mail systems including those in the Unix system have a wide range of basic facilities. Such facilities can be categorized as follows.

2.1.2.1. Creating Messages

An editing facility is required for creating messages.

The sophistication of this facility varies from system to system. Some systems only allow editing within the current line being input, while others, including Unix, provide more comprehensive word processing facilities with sophisticated features such as the ability to move paragraphs around.

2.1.2.2. Sending Messages

The capability to send a single message simultaneously to several people is an important feature provided by an electronic mailbox system. This is achieved by permitting the specification of multiple addresses. Some systems including Unix provide a more powerful version of this facility by providing copies of distribution lists, already filed away in the system, to be called up by using a short abbreviation. This is a powerful feature since it allows perhaps hundreds of copies to be sent by a user with no work beyond that required to send just one copy.

2.1.2.2. Receiving Messages

Most electronic mailbox systems provide a scanning facility for detecting waiting messages, and associated facilities that allow the user to select the order for accessing messages. Reply and forwarding facilities for assisting mailbox owners are also provided. The reply facility automatically completes the header information such as 'to' and 'subject' by copying the information from the message being read. Forwarding allows a message to be sent

on to another mailbox user along with any annotation that the sender wishes to add. Having read, replied to, or forwarded a message, the recipient can either delete, file or leave the message pending within his mailbox.

2.1.2.3. Filing and Retrieval

Some sort of message-filing capability is offered by most mailbox systems. However, the size of the store may vary from system to system, as may the index facility associated with the file. Search and retrieval of information held in these electronic files is usually undertaken using indexing classifications. The Unix mail system allows a user to retrieve information on message number, subject, and sender's id.

2.1.3. Mailbox System Structuring

More formal applications can be carried out via mailbox systems if additional facilities - mailbox structures - are provided. Mailbox structures refer to rules which organize mailbox messaging so that specific objectives for the communication can be met. The rules can be imbedded in the software and imposed automatically by a system. These can also be applied and policed by the user of the system. [1, 6]

Mailbox communication can be structured by organizing the information of messages [1, 6]. Software can sort communication into categories based on the content of messages and the interest groups with self-selected

memberships, for instance, or can permit recipients to route or filter messages by the information of messages such as keyword, subject, and author.

Structures can act as an aid to making decisions, getting agreement, and controlling work that is done via a mailbox system [1, 6]. Message summarizing or condensation can be accomplished by structuring the form of inputs. Senders might be required to adhere to length limitations, or to use votes or other numeric estimates instead of full messages. Summarizing can also be performed by human digesters who read incoming items, discard irrelevant ones, and summarize others before posting them.

In order to maintain ordered and useful communications, it may be necessary to control the access to the available facilities. A designated human leader or a software structure can help to perform these tasks: allocation and removal of mailboxes, allocation of basic facilities (e.g., 'read only' or 'read and write'), creation and control of distribution list or activities. Social pressure can also be used for this purpose - often group members collectively censure an errant member. If pen names are used or anonymity is maintained, individuals can vote to sanction or criticize errant members without embarrassing themselves. [1, 2, 4]

The QIRM system is based on "structuring communications by organizing the information of messages". Messages in the mailbox file can be organized into a database. This allows the retrieval of mail items via a user-freindly query language. It is an obvious advantage to be able to search

for messages using particular keys and conditions specified by the user. In the QIRM system, the user can query the retrieval system to find some specific information on sender's id, subject, sending date, receiving date, message number, status, recipient's id, and word or phrase in message body. In order to answer a query with proper information, the system imposes some structure on the messages. This structure is known to the system and used for the interpretation of the message.

2.2. QUERY LANGUAGES

2.2.1. Overview

One of the main objectives of organizing a large quantity of data in a computer storage is to be able to retrieve, modify, or insert any subset of the data. The user communicates his requests for information in the form of queries. The computer receives the queries, analyzes them, and proceeds to search and operate on the desired information. When a database is organized, considerable attention is given to the set of queries that will be directed at it. This enables the choice of the most suitable database organization for the query set. The study of queries play an important role in database organizations for information retrieval.

A query language provides a frame work for information retrieval. The query language is typically a high-level,

non-procedural language, in that the user only tells the system what information is required, rather than how it must be retrieved. The query language must also be complete. This implies that all legitimate data and data relationships are accessible through the operators defined in the query language. [19]

A query language is a generalization of the predicate calculus which is used to represent statements about the relation between attributes and values, or between two attributes which is used to identify a set (class) of objects [21]. The central feature is a quantifier function which is able to express, in a simple manner, the restrictions placed on a database-retrieval request by the user.

The formal query language contains three types of objects: designators, which name classes of objects in the database; propositions, which are formed from predicates with designators as arguments; and commands, which initiate actions [21]. Thus, in order to obtain the subject and sender's name of a message with status 'OLD', the following query might be formulated.

```
PRINT  Subject, Sender
WHERE  Status = 'OLD'
```

'Subject' and 'Sender' are designators to specify what information are needed for the answer; 'Status' and 'OLD' are designators which are arguments of the predicate "="; and "PRINT" is a command for specifying to the query analyzer the

form in which the resulting data should be presented or the manipulation which must be performed on the data [7].

2.2.2. Query Processing

After a query has been input by a user, it is worked on by a query processor. The flow diagram shown in Figure 2 demonstrates how a query processor interfaces with a data management system. The query processor receives a query from the user, and then parses and translates the query. The query processor utilizes information about the structure of the database known as the database description. This information is needed so that the parser can check the use of attributes of relations, if a comparison is made between attributes, and whether the domains are compatible, etc. The next step involves the determination of a schedule for processing the query. The set relationships among the data items that are required for answering the given query are determined during this stage. Optimization of the query may also be attempted at this stage, since the speed with which the query can be answered may depend on the choice made by the query processor concerning the sequence of steps to be taken by the system. The third step is the execution of the scheduled operations which involves actually searching the database and retrieving the desired data. The final step is report generation, which involves producing the desired output format for the specified data requested. [15]

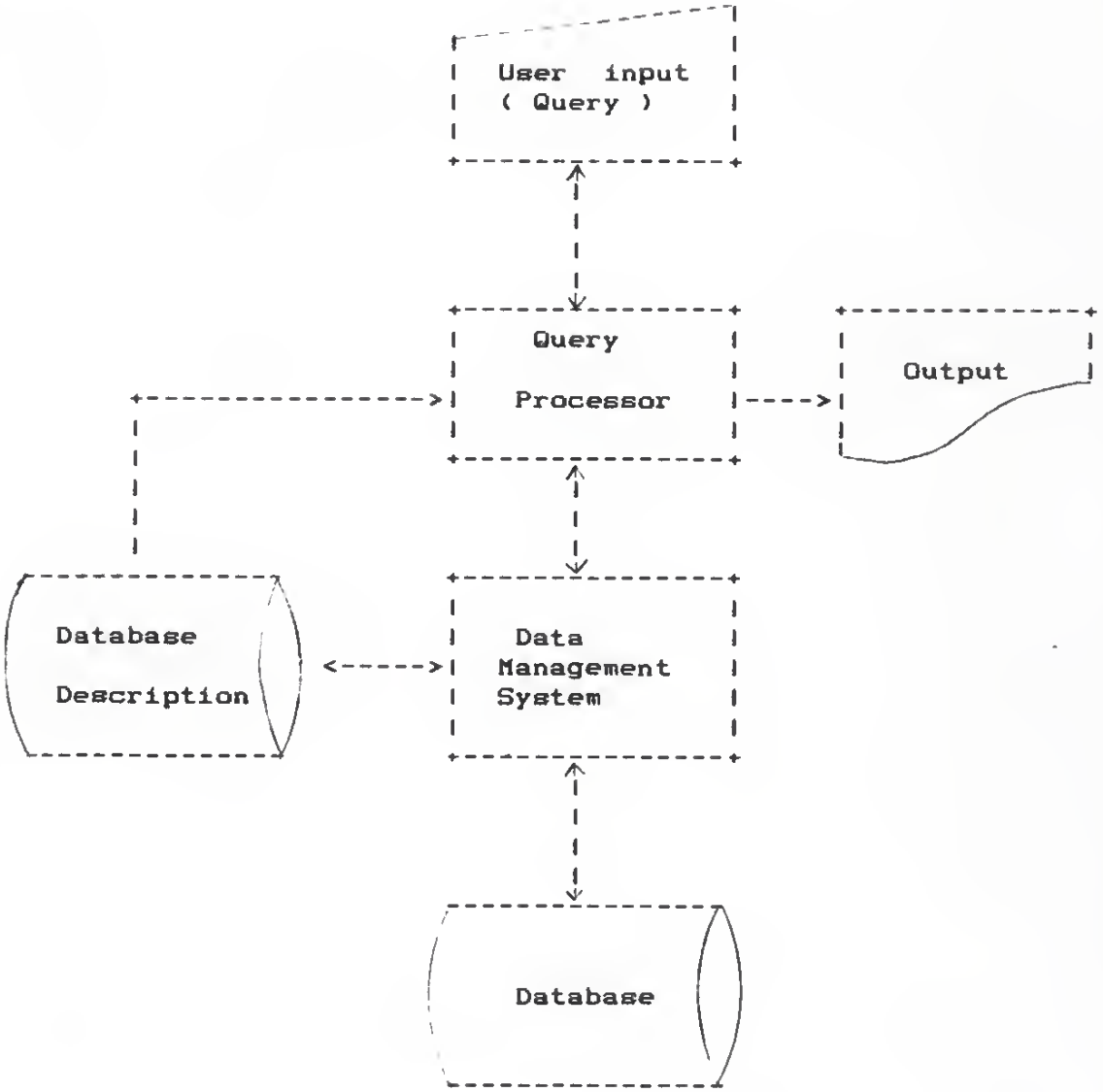

Figure 2. Query language system

2.2.3. Structured Query Language (SQL)

The most widely used query language is probably SQL [13]. SQL is based on the relational tuple calculus which is a notation for expressing the definition of a relation in terms of tuples belonging to some existing tables. Structured English Query Language (SEQUEL) is an older version of SQL which was developed by Chamberlain et al. of IBM as part of the SYSTEM R research project. Although it is termed as a query language, SQL permits updates as well as data definition. Its facilities are summarized in Figure 3 [14: p 110]. SQL can be used both as a stand-alone language and also as a data sublanguage embedded in PL/1 or COBOL. It is available in many well-known relational database management systems such as SYSTEM R, SQL/DS and DB2 [13].

Since data retrieval is the focus of interest of this report, this section reviews in detail only the query portion of SQL. In SQL the basic query construct is the SELECT-FROM-WHERE command. This construct forms the basis for retrieval.

Suppose that it is necessary to access the name of employee number 43 from the table EMPLOYEES. The appropriate command would be:

```
SELECT   ENAME
FROM     EMPLOYEES
WHERE    ENO = 43
```

The SELECT clause specifies the names of the fields (columns or attributes) that are to be selected - in this

Query Command	
SELECT	Retrieve data from one or more tables
Data Manipulation Commands	
INSERT	Adds one or more rows into existing table
UPDATE	Changes data in one or more rows of a table
DELETE	Removes one or more rows from a table
Data Definition Commands	
CREATE TABLE	Defines the structure of a new table to SQL
DROP TABLE	Removes the definition of the table from the system
ALTER TABLE	Adds a new column to a table definition
CREATE INDEX	Allows a table to be indexed on one or more columns
DROP INDEX	Removes the index from the system
CREATE VIEW	Defines a user view of part of the db
DROP VIEW	Removes the view from the system

Figure 3. Query, Manipulation, and Definition Commands in SQL

case one, but there can be several. The relation (table) to be used is listed after FROM. The WHERE clause contains a predicate which allows logical operators (NOT, AND, OR), standard comparison operators, IN, ALL and some other operators. The attributes in the predicate of the WHERE clause must be drawn from the tables of an appropriate FROM clause.

It is possible to specify just which columns are wanted. If all the details are needed an asterisk can be used:

```
SELECT *
FROM  EMPLOYEES
WHERE  ENO = 43
```

This query would give the whole record for employee 43.

Compound selections can be specified in the WHERE clause. More than one table can be used in the selection. The first scheme for doing this is to embed or nest a SELECT-FROM-WHERE command inside the WHERE clause, so that some column value(s) are matched with values selected from another table:

```
SELECT  columns chosen from table A
FROM  records in table A
WHERE  table A column =  SELECT table B column
                         FROM  records in table B
                         WHERE  condition
```

In effect, the query examines or extracts from table B a set of records which match the 'condition' specified in the nested WHERE clause. The value in some attribute column of table B in these selected table B records is then matched with a corresponding attribute in table A ("WHERE table A

column =*). Records from table A are selected where the match occurs. The final result contains the 'attributes chosen from this set of matching records'. The attribute chosen for matching must exist in both tables. It might be employee number for instance, or date.

SQL permits nested blocks to an arbitrary depth as long as the desired result (the answer) comes from a single relation. If, however, the result comes from two or more relations, the subquery strategy does not work. Consequently, it is necessary to join the relations together. This joining method is the second scheme of using two tables in a query.

The WHERE clause links the two tables by specifying the columns that are to be matched.

```
SELECT  columns (from table A & table B)
FROM  table A, table B
WHERE  column from A = column from B
```

This is an implicit join operation followed by a select. [12]

2.3. SEMANTIC NETS

The semantic net, developed by Quillian (1968) and others, has been proposed as an explicit psychological model of human associative memory. A semantic net consists of nodes, which represent objects, concepts, or events, and links between the nodes which represent the relations between the objects. A basic set of primitives is chosen to

describe objects and relationships, and all descriptions are constructed from these semantic primitives. The number and type of primitives that form the basic vocabulary is important because the choice of primitives will determine the expressive power of the representation.

Let us suppose that we want to represent the fact that 'All poodles are dogs.' in a semantic network. We can do this by creating a simple net in which the nodes represent the objects and the link, the 'is-a' (or 'subset') relation between them (Figure 4).

If 'Benji' is a particular poodle, and we want to add the fact that dogs have tails, then we would add two nodes and two links, one of the new links is a 'has-part' relation (Figure 5). This representation enables us to deduce facts that are not explicitly stated in the network, e.g. that poodles have tails since they are dogs, and so does Benji since he is a poodle. This feature is called property inheritance. Care must be taken to separate generic concepts (or objects), such as poodles from a specific token such as Benji, otherwise errors in deductions can result. Linking can result in incorrect deductions if the generic and specific nodes are intermingled, or if the inheritance characteristics are not carefully isolated. [9]

The semantic network representation is not a formal mathematical system with unifying principles. Its use tends to be rather ad hoc, with various researchers employing different net interpretation schemes based on the same general concepts. [21]

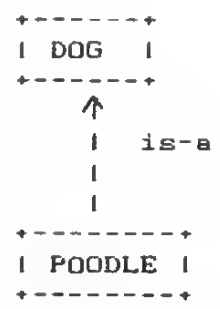

Figure 4. A Simple Semantic Net

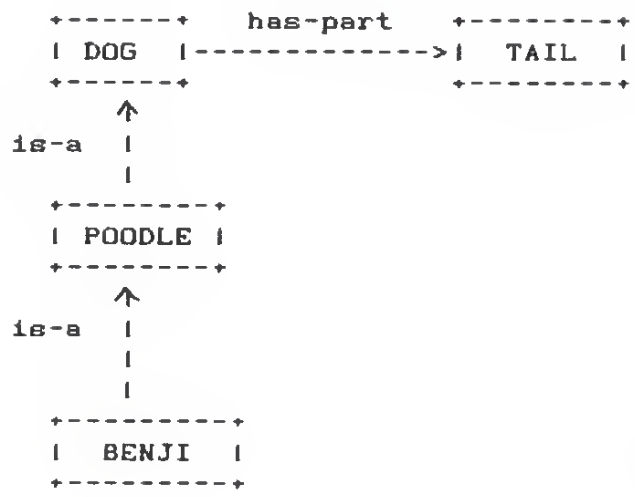

Figure 5. A Semantic net using IS-A link

2.4. SEMANTIC NETWORKS, RELATIONAL DATABASES, AND QUERY LANGUAGES

Database management systems typically incorporate a database schema. The schema is actually expressed in terms of a descriptive language called a data model. The data model provides a set of constructs for describing aspects of the database and is used by the database management system for processing queries.

The semantic network model and the relational data model could be used both for describing the data and for specifying queries. In processing queries using the former scheme, special algorithms can be employed to match the graph corresponding to the query with the graph representing the data [21]. For the case involving relational notation, most queries can be viewed as taking an entity that meets certain criteria, connecting it to an entity of another type - perhaps through many relationships, and finally returning some attributes of the resulting entity.

2.5. FRANZ LISP

The language used for implementing the QIRM system is Franz Lisp, a popular dialect of Lisp. The Lisp language which was invented in the late 1950s has evolved in a number of different directions. Consequently, unlike many other languages, there is no such thing as standard Lisp. Among the most widely used dialects of Lisp today are MacLisp: a version of Lisp developed at MIT, and InterLisp: developed

at Bolt, Beranek and Newman, and Xerox Palo Alto Research Center. [3]

Franz Lisp was developed at the University of California at Berkeley and is available under Berkeley Unix [3]. Its roots are in a PDP-11 Lisp system which originally came from Harvard. As it grew it adopted features of MacLisp and Lisp Machine Lisp. Substantial compatibility with other Lisp dialects (Interlisp, UCIlisp, CMUlisp) is achieved by means of support packages and compiler switches. The Franz Lisp system consists of an interpreter, Lisp, and an optimizing compiler which is named Liszt. The kernel of Franz Lisp is written almost entirely in the progrmming language C, with much of the support written in (compiled) Lisp. For run-time efficiency, small portions of the kernel are written in assembly language. [16]

Franz Lisp is capable of running large Lisp programs in a timesharing environment. It has facilities for arrays and user defined structures, along with a user controlled reader with character and word macro capabilities which gives the Lisp programmer the ability to modify the way expressions are read in by the interpreter. Through the use of read macro, the user can designate special characters which act in unusual ways. This gives the user the ability to establish useful shorthand that simplify some programming tasks. Also, Franz Lisp can interact directly with compiled Lisp, C, Fortran, and Pascal code.

2.6. RELATED WORK

Much of the work which has been done on structuring electronic mailbox communication involves quite different approaches. Some articles which are representative of these different methodologies include [2], [5], and [4].

The followings describes portions of these works.

2.6.1. Systems for Controlling Group-Communication (SCGC) [2]

It is often easy to send a message to a large number of people since systems are often designed to give the sender too much control of the communication process, and the receiver too little control. If the receiver is given more control over the communication process, much of the electronic mail which is not of interest to a person can be greatly reduced. In order to accomplish this, a structure must be imposed on the set of messages. Electronic mail systems thus need to be more database oriented, like some of the existing computer conference systems. Even though group-communication system can cause information-overload problems, it provides people an environment for meeting and exchanging ideas much more freely than in a pure mail system. There exist several design options for electronic message systems to overcome the overload problems.

When a conference system complements a message system, the flow of unwanted messages is greatly reduced. Instead of delivering an unordered heap of messages, the system can deliver a neatly structured database of incoming messages.

It permits the reader to decide which messages to read immediately, which to save for another time, and which not to read at all.

Another way of structuring messages is via comment trees. A system can be designed to store relations between messages, where one of them can be a comment or a reply to another message. Thus, a set of messages which refer to each other directly or indirectly (comment tree) can be identified automatically by the system.

Also, the storage and retrieval of messages can be controlled by such items as: keywords, subject, and author. The system can be told to deliver messages according to user-specifications involving these items, thus giving the reader more control of what messages will be read. For example, a user can read all messages on a certain subject before continuing with a new subject.

A designated human leader for a computerized conference can help a group to control its communication. A leader's role includes editing a list of items or keywords for clarity, or deleting or moving inappropriate items before posting them (control by selection). The process of summarizing discussions can also be performed by human digesters. People in such roles abstract the discussions in voluminous open conferences into write-protected conferences containing only the abstracts (control by abstract writing).

2.6.2. A System for Managing Structured Messages (SMSM) [5]

Message systems transport the messages, but do not manage them. Database management systems manage the information, but do not have any notion of addresses. Integration of the facilities of both systems provides a scheme for structuring mailbox communication.

A system using such an approach manages messages as typed objects which can be stored within a logical unit and transferred between the units. Such a system provides the manual functions which enables users to find and query messages by selecting a message type and partially specifying the contents of the messages in templates. A user can specify message selection based on various combinations of items internal to the messages.

Also, the system permits the specification of procedures which are triggered by the presence of messages and which automatically manipulate the messages. The automatic procedures are specified by giving the system some indication of the pattern or contents of the messages which are desired and an indication of what the system should do with these messages. Automatic procedures run regardless of whether the user who specified the procedure is currently logged in. Examples of these automatic functions include: coordination of messages, i.e., act only when a related set of messages has been assembled; modification and creation of messages; filing messages; and forwarding received messages to other stations according to their contents.

A uniform user interface which is based on 'specification by example' can be provided to carry out all the manual and automatic functions. Users query messages by partially specify the contents of the messages in the template. The automatic procedures are also specified by indicating what messages are to be collected, and what is to be done with them.

2.6.3. Intelligent Information-Sharing Systems [4]

The problem of balancing the value of open communication channels with the cost of information overload has been expressed by many users of group-communication systems. A technology that can increase the selectivity with which the information is disseminated should be sought. A prototype called Information Lens (IL) has been developed using this concept. This system employs user-interface design and techniques from Artificial Intelligence such as frames, production rules, and inheritance. These techniques help people filter, sort, and prioritize messages that are addressed to them. They also help users to find useful message they would not otherwise have received, via a special mailbox called 'Anyone'. Messages that have 'Anyone' as an addressee are automatically delivered to a public mailbox. Receivers can have interest profiles which automatically retrieve messages from the public mailbox 'Anyone'. Also, this system permits semi-structured templates to be used by senders in message composition. These templates can

also be utilized by recipients to facilitate construction of a set of rules for filtering and categorizing messages.

Three different approaches to automated message filtering are employed in the Information Lens system. A cognitive filtering approach works by characterizing the contents of a message and the information needs of potential message recipients and then using these representations to intelligently match messages to receivers. Decisions are based on distribution lists, and either a simple keyword search or combinations of various conditions on fields. A social filtering approach works by supporting the personal and organizational interrelationships of individuals in a community. It complements the cognitive approach by focusing on the characteristics of a message's sender, in addition to its topic. An economic filtering approach relies on various kinds of cost-benefit assessments and explicit or implicit pricing mechanisms. The length of a message, the number of recipients, and the salary of a recipient are some of the factors used to estimate its cost. The current version of this system emphasizes the cognitive approach.

The Information Lens system is written in the Interlisp-D programming environment using Loops, and runs on Xerox 1108 and 1109 processors connected by an Ethernet. It is built on top of an existing electronic mail system. Users can continue to send and receive their mail as usual, and have the option of using centrally maintained distribution lists and manually classifying messages into folders. The system additionally provides following important optional

capabilities: (1) Structured message templates are available for message composition; (2) Senders can include a special mailbox named "Anyone" (which is a public information file as) an addressee of a message; (3) Receivers can specify rules to automatically filter and classify messages arriving in their mailbox or the "Anyone" mailbox. Rules can move messages to folders, delete messages, set "characteristics" of messages based on other field values, or select messages addressed to "Anyone".

2.6.4. Comparison of QIRM with related work

In the articles [2], [5], and [4], the authors presented their ideas on desirable design options and implementation strategies for structured mailbox systems. Some considerations in formulating these strategies includes: information overload [2, 4], information sharing [2, 5, 4], communication filtering, sorting, and prioritizing [2, 5, 4], the query service [5, 4], and intelligent database management systems [5, 4].

The features presented in [2], [5], [4], and the QIRM could be compared in many aspects.

The scope of QIRM is different from that of the systems illustrated in [2] (SCGC), [5] (SMSM), and [4] (IL). QIRM is a information-retrieval system which is intended to access messages in a user's mailbox. SCGC, SMSM, and IL are group-communication-oriented mailbox systems for sharing information. In the environments of SCGC, SMSM, and IL,

users can send, receive, delete, file messages as well as retrieve them. The recipient using SCGC, SMSM, and IL can query both global and local messages. In the QIRM, however, a mail-receiver can query only messages that have arrived in his own mailbox.

The QIRM differs from SMSM and IL in the methods of specifying the queries. While the queries of SMSM and IL are based on the use of semi-structured message templates (a domain- calculus query), QIRM's query is specified in the way similar to that of SQL (a tuple-calculus query). It should be mentioned that IL provides very friendly and convenient user-interfaces. Messages in IL are composed with a display-oriented editor and templates that have pop-up menus associated with the template fields.

SCGC, SMSM and IL provide both automatic and manual facilities for structuring messages, while QIRM has a manual function that categorizes messages temporarily according to the user-query. Structuring on the message set is based on information input by: the sender(SMSM, SCGC, IL), someone else (e.g. a leader) (SCGC), the recipeint (SMSM, SCGC, IL, QIRM), or automatically the system (SMSM, SCGC, IL). SMSM and IL allow a recipient to specify rules for processing messages. These rules are composed using the same templates as those used for composing and quering messages. The facility for user-specified rules is not employed in QIRM nor SCGC.

Both QIRM and IL use several techniques from artificial

intelligence. To structure messages, the latter employes frames, each of which contains messages with similar contents. These frames are arranged into a network using the frame-inheritance lattice. The messages in QIRM are structured in a semantic network that consists of many subsets, and each subset represents a message.

Even though the detailed architecture and techniques employed in these systems are different each other, the basic key ideas are found to be very similar: Messages can be controlled and selected on the content of messages (e.g., author, topic, keyword) by a user's message-specification or by the system function which automatically manipulate the messages; In order to achieve such a structured communucation, it is desirable to develop more database-oriented and active mailbox systems.

CHAPTER 3

DESIGN

3.1. OBJECTIVES

The objective of this work is to design and implement a system which allows a user to categorize requests for messages in his/her mailbox and to retrieve the information according to the fields and conditions specified by the user. By employing concepts and techniques of artificial intelligence, the system can provide some insights for developing an intelligent user-friendly UNIX mail facility. Moreover, the project has the potential of reducing many of the burdens and problems that current users of Unix mail have to encounter during their use of this facility.

3.2. THE SYSTEM ENVIRONMENT

The prototype which is referred to as 'The QIRM System' has been developed on a Digital Equipment Corporation VAX 11/780 minicomputer supported by the Berkeley 4.3 BSD UNIX operating system at Kansas State University and written in Franz Lisp Opus 42.

3.3. SYSTEM DESIGN

3.3.1. Overview

The diagram shown in Figure 6 presents the framework of the QIRM system. As can be seen, the system consists of four major components. The following is a description of each of these basic components.

(1). DATABASE PREPARATION : Database preparation involves the design of how the data are organized to be used and the subsequent loading of data into the database from the input file(s).

(2). USER INTERFACE : The user interacts with the system through a query language. The entire retrieval design process is accomplished through this interface.

(3). QUERY PROCESSOR : This processor processes a query submitted by a user. The query is parsed, and the database is searched for items that match the specific request. Information about the organization of database is incorporated within this processor itself.

(4). OUTPUT : The output, which is the result of the searching operation performed by the query processor, is selectively generated according to the user's query-specification.

Processing by the QIRM system can be broadly divided into four phases. In phase one the database is constructed; the data read from the data file(s) are converted into the desired Lisp structure. The next phase is query processing which involves scanning, and parsing the query submitted by a

user. The third phase involves searching the database and retrieving the requested data. The final phase is the output generation. In this phase, the requested information is displayed on the terminal in the format specified by the user.

Having provided a basic background of the QIRM system, in the following text each component is described in considerable detail.

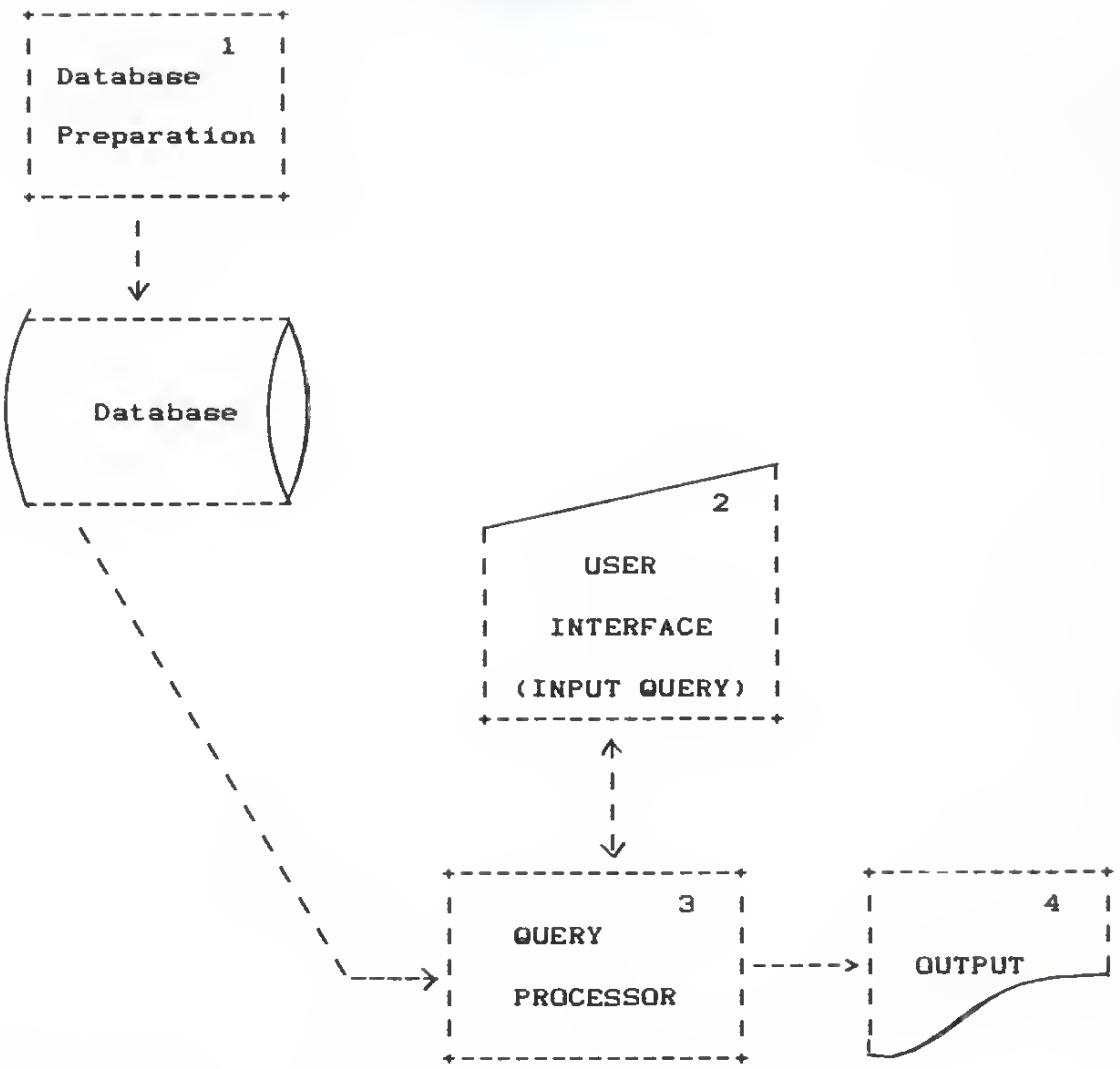

FIGURE 6. Framework of the QIRM SYSTEM

3.3.2. Database Preparation

3.3.2.1. The Lisp Notation and Indexing Scheme

The database in this system is based on the semantic network formalism (See section 2.3.). The semantic networks are augumented with an indexing scheme.

As expressed in Section 2.1.1., the contents of a message in a Unix system mailbox consist of a header with various fields and a text body. This structure can be used for storing mailbox information into the database. Figure 7 presents a graphical description of the information that is used to structure a database in QIRM. Each message has seven properties which are 'sdate, rdate, from, to, subject, status, body'. A brief description of each field is now given:

'sdate' indicates the date of sending a message;

'rdate' indicates the date of receiving a message;

'from' indicates the message-sender's id-name;

'to' indicates the message-recipient's id-name;

'subject' indicates the subject of a message;

'status' indicates the status of a message such as 'Has this message been read or not?' or 'Is this message old or new?';

'body' indicates the body of a message.

Body1 is a node with two links, one to a node containing the text portion of body, and the other to a node containing control information about a message, namely, the number of lines of text and the address of text in the mailbox file. Rdate1 is a node having four properties which are 'rd, rw,

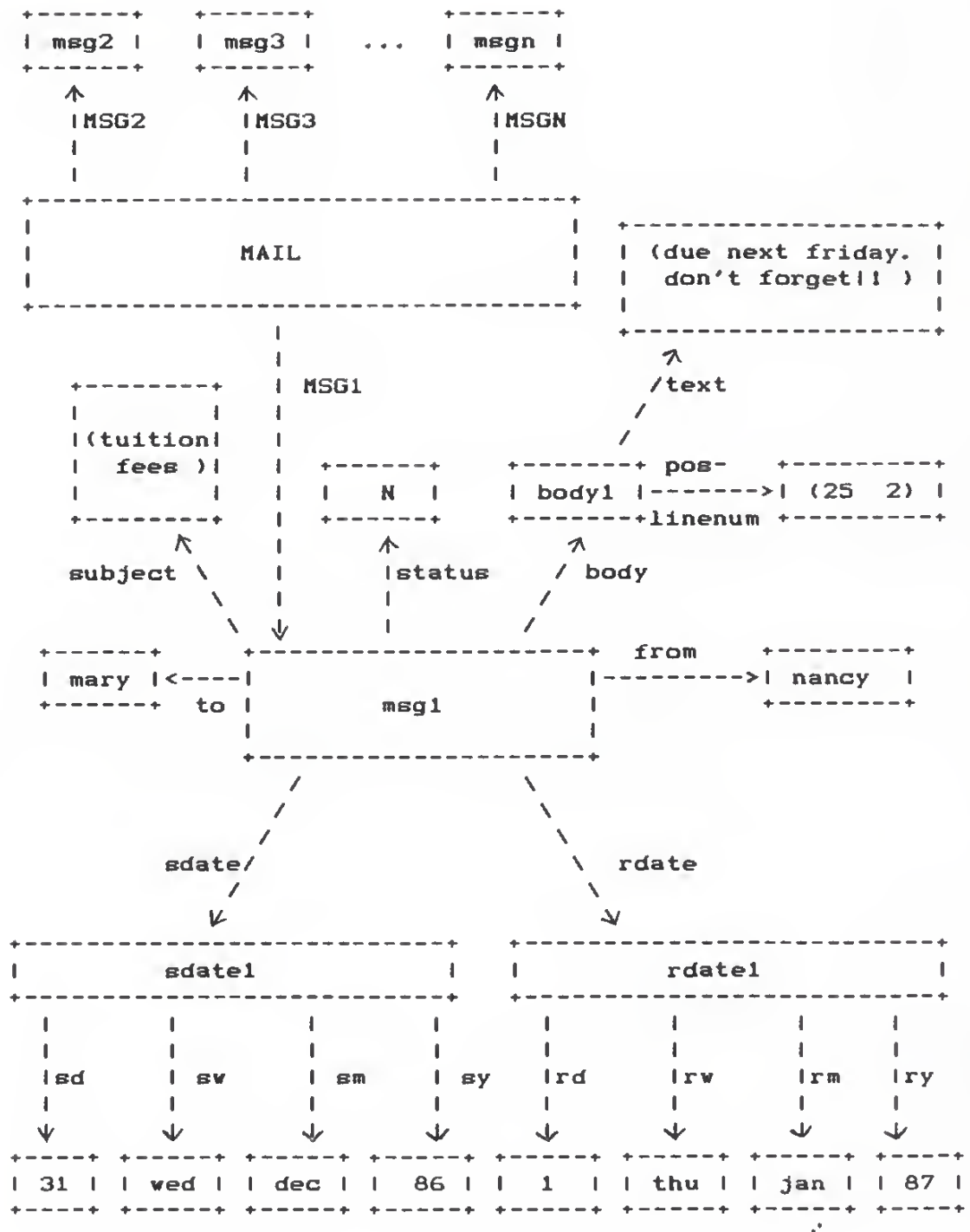

Figure 7. A semantic network showing the structure of the database in the QIRM system

rm, ry'. The property links 'rd, rw, rm, ry' indicate the day number, the weekday, the month, and the year of receiving a message, respectively. Sdate1 is a node containing four properties which are 'sd, sw, sm, sy'. The property links 'sd, sw, sm, sy' indicate the day number, the weekday, the month, the year of sending a message, respectively.

The format of the semantic network shown in Figure 7 suggests that the information about each of the individual messages, like msg1, is clustered in a particular place. Any fact that is associated with msg1 is represented with an arrow going in and out of the msg1 node. Therefore, having located msg1, it is possible to gain access to all the information about it. This is an indexing scheme. If a certain process in program requires information about the subject of msg1, it would not be practical to linearly search all the nodes in the database to find the fact. It is much more plausible to have msg1 point to the information directly.

There are a variety of techniques that have been developed for indexing patterns in a database. The technique employed in this work takes advantage of Lisp property lists in breaking up a large database into several small ones. As shown in Figure 7, the database 'MAIL' is composed of many records such as 'msg1, msg2, ..., msgn'. In order for a database to support several records at the same time, there must be an index which keeps track of the records. The capitalized record name (i.e. if a record name is 'msg1', 'MSG1' becomes its capitalized name.) can be used as the name

of the property to index a record under. Thus, when a record is added to the database, it is stored on a 'capitalized-record-name' property list of 'MAIL'. When an item is added to a record in the database, the system identifies the record (a message-name) the item belongs to and stores it on a list under some property name of the record (a message). For example, suppose the system wants to add an information such as 'The message subject is TUITION FEES.' to a record, say msg1, using this scheme. It can be done by adding this item to the list stored under the 'subject' property of a record 'msg1' which is stored under the 'MSG1' property of the database 'MAIL'. When fetching an information like the subject of message1 from the database, the system first obtains the list under 'MSG1' property of 'MAIL'. Then, it gets the value under 'subject' property in this list.

From the above discussion, we can conclude that: Semantic networks suggest a scheme of forward and backward pointers that appears to make accessing information very easy. Figure 8 shows how the attribute-value memory structures in Figure 7 are represented in Lisp.

```
   ATOM                    PROPERTY  LIST
-----------          ------------------------------

  MAIL                ( (MSG1    msg1)
                        (MSG2    msg2)
                        (MSG3    msg3)
                              :
                              :
                        (MSGN    msgn)                          )

  msg1                ( (subject    (tuition fees) )
                        (status      N )
                        (to          nancy )
                        (from        mary )
                        (body        body1 )
                        (rdate       rdate1)
                        (sdate       sdate1) )

rdate1                 ( (rd    1 )
                         (rw    thu )
                         (rm    jan )
                         (ry    87 )   )

sdate1                 ( (sd     31 )
                         (sw     wed )
                         (sm     dec )
                         (sy      86)   )

body1                  ( (text    (due next friday.
                                      don't forget!! ) )
                         (pos-linenum   ( 25   2 ))          )

   :                                  :
   :                                  :
   :                                  :

msgn                   (     (        :
                                      :
                                      :
                                      :
                                      :                  ))


                                                         ;
```

Figure 8. The Lisp Representation of the Database Shown in Figure 7

3.3.2.2. Database Loading

After the data structure is defined, the database is loaded by the database manager; this manager consists of several procedures. The database loading process involves determining the source of the data, reading the data file, constructing the database, and updating database file. The question of which data file(s) should be loaded is critical in terms of efficiency.

As shown in Figure 9, the QIRM system has been designed to take one of two paths in determining which input file to read. Which path is chosen by the database manager depends upon the existence of an old mail database file (referred to as 'old-db'). The old mail database file is arrived at by concatenating the user's login-id and the string 'data'. For example, if 'songhee' is the user's login-id, then 'old-db' file is named 'songheedata'. That is, the database in 'songheedata' is the one that had been constructed when QIRM was called the last time by a user, 'songhee'.

When a user invokes the QIRM system, the database manager checks whether an 'old-db' exists or not. If there is no 'old-db' found, the database manager considers that it is the first time that the user has employed the QIRM system. In this case, the manager takes the user's mailbox as the only data file and loads the data from the mailbox into a database ('path two' in Figure 9). However, if 'old-db' is found, the manager uses both an 'old-db' and a mailbox as data files to reload data into a database. The reloading process

includes the reading of the two data files to compare a key portion of each message in one file with that in another file. This comparison is needed for the manager to identify the messages deleted from, or added to the most recent mailbox ('path one' in Figure 9). Using the algorithm (algorithm A) described above, the database manager constructs a database and stores it into 'old-db', which will be used as one of the input files for the next usage.

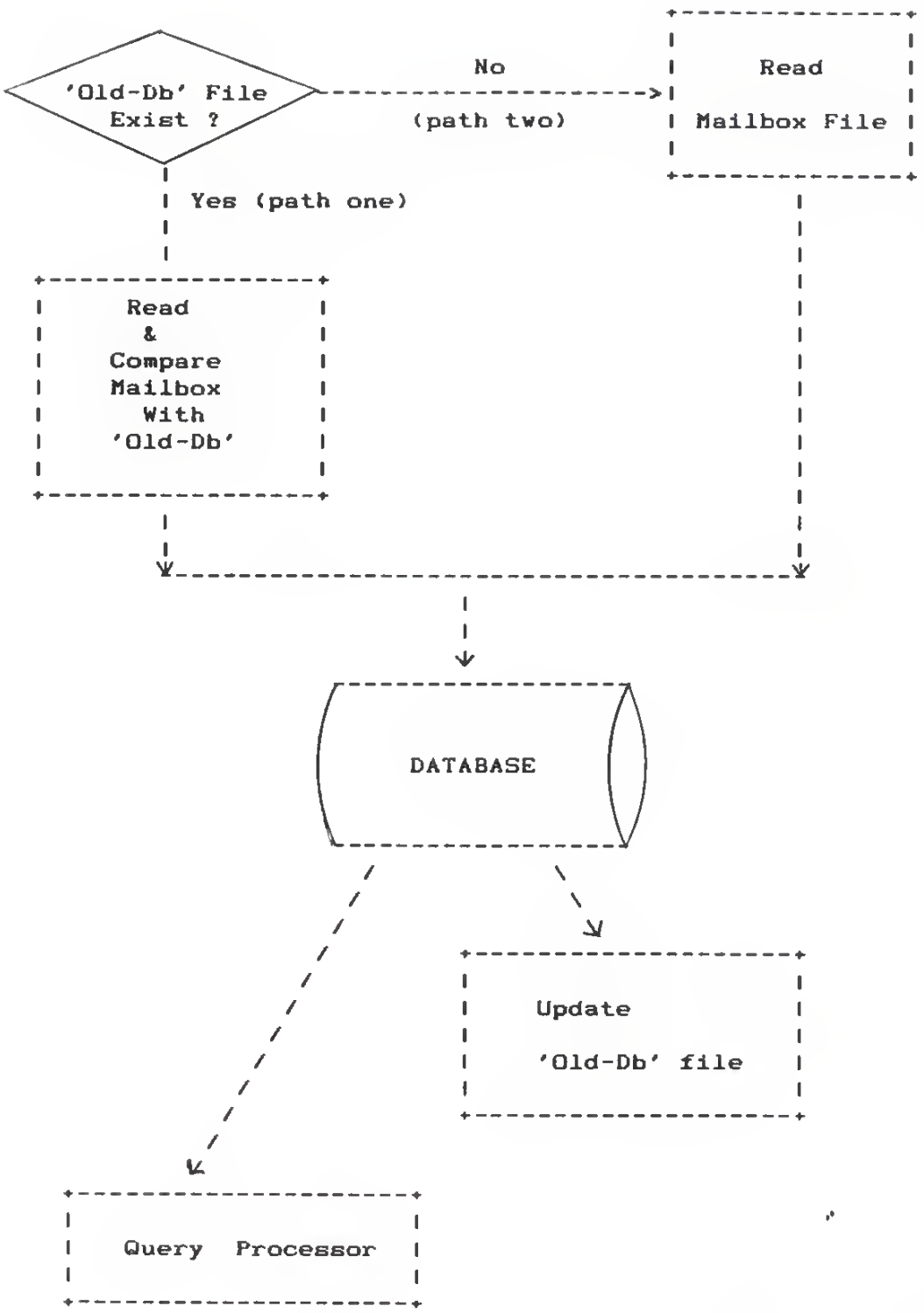

Figure 9. Algorithm A for loading a database

3.3.2.3. Efficiency Consideration

For the purpose of efficient database loading process, two algorithms (A and B) are compared by considering the time complexity. Algorithm A was discussed in the previous section and shown in Figure 9, while algorithm B is illustrated in Figure 10.

In this argorithm B, a user's mailbox is taken as the only data file. Whenever the system is invoked, a database manager reads through a mailbox and constructs a database. No physical database file exists in this algorithm.

Algorithm A uses several procedures to minimize the regular user's waiting time for database-loading, by distinguishing the first-time user from the regular user of the system. In order for the text data in a mailbox to be loaded into a database, the data should be organized and converted to the desired format. But, such processes are not required for loading the data from 'old-db' which have been stored in the lisp format consistent with the database structure. Therefore, once the database manager finds that the key portion of a message, say msgN, in 'old-db' is the same as that in mailbox, the whole information about msgN in 'old-db' can be reloaded into a database quickly and easily. Algorithm A takes extra time for updating 'old-db' and determining the identity of messages in two data files. In order to minimize the time for the latter, this algorithm uses very small key-portions of messages for the comparison.

The performance of each algorithm was evaluated by

running it with a set of sample data which were saved into a mailbox 'songhee' from the 14th of Jan 1987 to the 20th of July 1987, and by measuring the user time which is needed to load the whole data from the data file(s) into a database. The data set includes various messages such as the local messages (sent-received on ksuvax1), UUCP messages, and the messages from CSNET, BITNET, KSUVM, JUNET via CSNET, ARPANET, and USENET. In each comparison, the key portion (i.e. the first line of a message header containing a sender's id and the date of receiving the message) of the last five messages were modified for algorithm A, so that the system recognizes them as new messages.

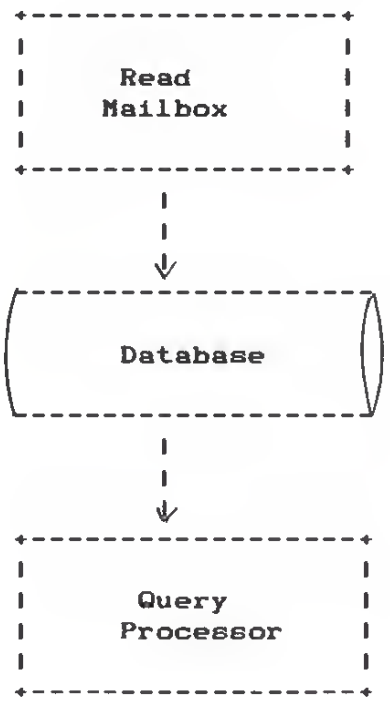

Figure 10. Algorithm B for loading a database

For the purpose of predicting the loading times for large sets of messages, the predicted loading times, T(second), for each algorithm were formulated by utilizing linear regression method. Minimization of the sum of squares of the deviations between the measured times and the linear expression yields:

$$T = 1.72 + 0.33N \quad \text{--- Algorithm A}$$

$$T = -3.40 + 0.39N \quad \text{--- Algorithm B}$$

where N denotes the number of messages. The correlation coefficients for algorithm A and B were 0.997 and 0.999, respectively, which represent good linearities of the measured loading times in both cases.

As shown in Figure 11, algorithm A seems to be less efficient than algorithm B for the small number (under 50) of messages, but the difference is negligible. The time used for database loading in both algorithms are almost identical for fifty to one hundred messages. For a large set of data (more than 100 messages), however, algorithm A becomes more efficient than algorithm B. The larger the data set, the more loading time is saved by using algorithm A. For example, a user with 360 messages can save more than 15% of loading time which is required by algorithm B.

When we consider the complexity in terms of space, algorithm B is superior to algorithm A. The space for 'old-db' is not needed in algorithm B. Also, the source code for algorithm B is shorter than that of algorithm A by 2937 bytes. However, for the QIRM with a heavy and direct user-interface, it is obvious that the time efficiency is a

much more critical factor than space efficiency in choosing an algorithm. For this reason, algoritm A has been chosen for loading a database in QIRM.

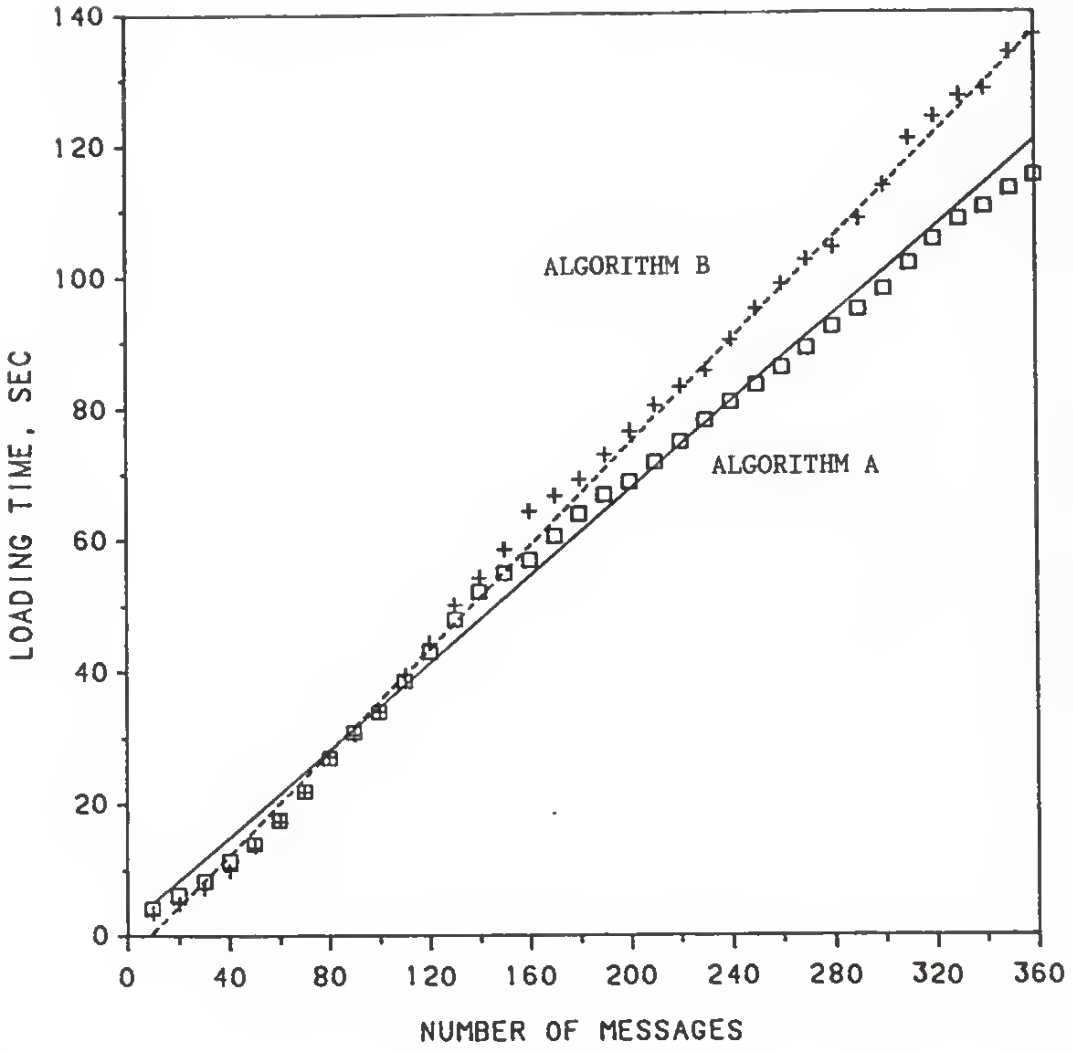

Figure 11. Comparison of loading time of algorithm A with that of algorithm B.

3.3.3. Queries - User Input and Output

As mentioned in chapter 1, the query language is very useful for retrieving information from data bases. Even though the query language is very different from Lisp, it is convenient to describe the query language in terms of the general framework of Lisp. It is described as a collection of primitive elements, together with means of combination that enable a user to combine simple elements to create more complex elements and provides a means of abstraction that enables users to regard complex elements as single conceptual units. The mail-query language implemented in this project has been designed taking advantage of above aspect in Lisp.

In order to illustrate the features of the query system in the QIRM, this section shows how QIRM can be used to manage the database which is built from the information in a mailbox. The language provides pattern-directed access to the information.

3.3.3.1. Simple Queries

The mail-query language allows users to retrieve information from the database by posing queries in response to the system's SELECT-WHERE prompts.

The syntax of simple query is

SELECT (<field>+ | *)

WHERE (<query-pattern>) | ()

Some points concerning this query are:

1. a. <field>+ indicates the set of length one or more which consists of elements of the form <field> where

   <field>, in turn, is made up of sdate, rdate, from, to, subject, status, and body.

   b. '|' indicates alteration, for example, 'A|B' means a choice of A and B.

   c. '*' is the shorthand of all fields.

2.

2.1. <query-pattern> has the following structure.

<predicate-operator> <propertyname> <propertyvalue>

where

a. <predicate-operator> can be <, >, <=, >=, or =.

b. <propertyname> can consist of sw, sd, sm, sy, rw, rd, rm, ry, from, to, subject, status, and text.

c. 1. <propertyvalue> can be the value to be searched for in the named property of messages.

2. If the property value consists of more than one word, it should be parenthesized, otherwise, it should not. A word indicates a sequence of any characters except blank.

2.2. () indicates that 'no condition' is specified for retrieving messages.

The input query specifies that one is looking for entries in the database that match a certain pattern. Using the matching operation that will be described in section 3.3.4.1., the query determines whether the desired pattern is in the database and which records of the database contain it. The system responds to a simple query by showing the values of requested fields in all records found which meet the criteria specified by the pattern.

A query's response involves the output of data. Sometimes the amount of data is very large, and it may be unexpected by the user. In such a case, it is convenient to tell the user the number of messages retrieved and output only the first items, and then inform the user that there are more data which can be supplied. The system gives the number of additional reponse records and interrogates the user about their disposition.

For example, to see all subjects of mail items which were received on Wednesday, one can say

```
SELECT  (subject)
WHERE   (=  rw  wed).
```

The QIRM system does not distinguish between upper and lower case letters. Thus the above query could equally well have been entered as

```
SELECT  (SUBJECT)
WHERE   (= rw WED).
```

The system would display the subjects of all the records on a screen having the structures which satisfy the condition of

'WHERE' clause. Figure 12 shows an example of how the output is displayed at the terminal and how the system interacts with a user.

```
     **  3 message recalled!   **
     msg4
     subject:  new arrival

     2 messages left!    More? (y/n) n

     SELECT  (subject)
     WHERE   (>  rw  wed)

     **  1 message recalled!  **
     msg1
     subject :  tuition fees

     SELECT  stop

     **  bye bye  **
     %
```

Figure 12. Output Examples of QIRM

During retrieval of information, the user may abort the display of the remaining messages, and start on a new query. That is, by choosing 'n' as the answer to the system's 'More? (y/n)' question, the user can start a new dialogue. The user can either continue retrieving information by specifying his choice of fields and conditions in response to the system's new SELECT-WHERE prompts or exit the system by SELECTing 'stop' command.

Another example is:

```
SELECT (sdate body)
WHERE  (= subject  (A.C.M. meeting) )
```

The system would respond with the selected items from messages which meet the conditions specified by the 'WHERE' clause.

In this example, one of the output might be :

```
"  msg10
     Date: Wed, 14, Jan 87 08:45:04 cst
     ( ...  the message body of A.C.M. meeting  ... ) "
```

If one wants to retrieve all the values of seven fields of messages, an asterisk can be used:

```
SELECT  (*)
WHERE   (=  subject meeting)                    "
```

would give all of fields of each mail item whose subject contains the word 'meeting'. Notice that a user can specify either the exact subject of a message or a keyword of the

subject.

Thus, the simplest form of query which will display all of the messages in the database in their entirety is:

```
SELECT  (*)
WHERE   ()
```

3.3.3.2. Compound Queries

Simple queries form the primitive operations of the mail-query language. In order to form compound operation, this query language provides a means of combination which mirror the formulation of logical expressions. Here, logical connectives, 'and' 'or' and 'not', could be considered as operations built into the query language.

The syntax of a compound query with 'and' is

```
SELECT  ( <field>+ ) | (*)
WHERE   (and  <compound-query-pattern>++ )
```

Some points concerning this query are:

1. <compound-query-pattern>++ indicates the set consisting of two or more <compound-query-pattern>s.

2. <compound-query-pattern> can be any of the following:

```
(and   <compound-query-pattern>++  )
(or    <compound-query-pattern>++  )
(not ( <query-pattern> ) )
( <query-pattern> )
```

The compound query with 'and' connective is satisfied by all sets of values for the property names that simultaneously satisfy all of <compound-query-pattern>s.

The following is a compound query example which illustrates the use of 'and':

```
SELECT  (subject)
WHERE  (and ( >= sm feb )  (= to grads) )
```

As the response of this query, the system shows all the subjects of messages which were sent to the graduate students from Feburary to December.

Another means of constructing compound queries is through 'or'. The syntax of this query is:

```
SELECT  (  <field>+ )  |  (*)
WHERE   (or  <compound-query-pattern>++ )
```

This query is satisfied by all sets of values for the property names that satisfy at least one of <compound-query-pattern>s.

An example of a compound query using 'or' is:

```
SELECT  (body)
WHERE    (or (= text (unix system))  (=  text   (mail
          box)))
```

The result of this query is all the message bodies which contain either the word 'unix system' or 'mail box'.

Queries can also be formed with 'not'.

SELECT (<field>+) | (*)

WHERE (not (<query-pattern>))

is satisfied by all values of the property name that do not satisfy query-pattern.

A query example formed with 'not' is :

```
SELECT  (from)
WHERE   (not (= sm  jan ))
```

The result of this query is senders' id-names of all the messages which were not sent in January.

Also, one can combine 'and', 'or', and 'not' to specify conditions in a WHERE clause as shown in the following example:

```
SELECT  (subject  body)
WHERE    (or  (and  (= sd 20)  (not(= sm jan)))
              (= status r))
```

As the result of this query, the system shows a user the subjects and text bodies of all the messages which either were sent on the 20th of any month except January or have already been read by the user.

3.3.4. The Query Processor

In this section, an overview of the query processor's general structure is presented.

The query processor is organized around a central operation called network-fragment matching. This section begins by discussing network-fragment matching and how it permits both simple and compound queries to be implemented. This section also shows how the entire query interpreter works by utilizing a function which classifies expressions.

3.3.4.1. Matching

The mechanism used by this processor is based on matching semantic network structures: a fragment of a semantic net is structured to represent an object (a query-pattern which is sought). This fragment is matched against the database for the semantic net to see if such an object exists. Once having found the object, variable nodes in the fragment are bound to the values which they must possess in order to make a match perfect.

Suppose we wish to retrieve the information based on the following request indicating 'Show all subjects of messages whose status is 'New':

```
SELECT  ( subject )
WHERE   ( = status N )
```

We construct the fragment as shown in Figure 13.

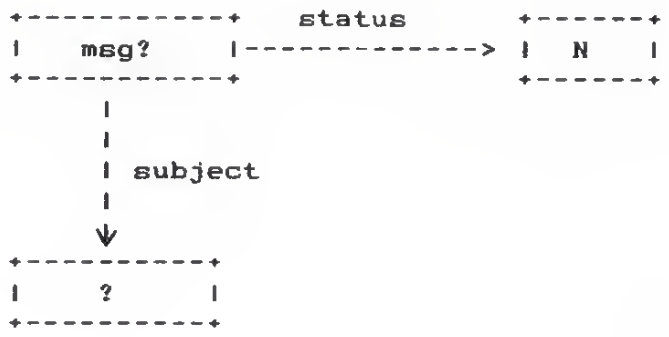

Figure 13. A Fragment of a Semantic Nets

This fragment is then matched against the database in a search for a 'msg?' node that is connected to a node containing 'N' by a 'status' link. If it is found, the node to which the SELECTed field link (a subject link) points is bound in a partial match and might be used to formulate a query response such as:

'msg1
subject : tuition fees'.

Had no match been found, the answer would have been

'No message whose status is 'N' is found'.

3.3.4.2. Stream Implementation

The testing of query patterns against network fragments utilizes the notion of streams. A stream is simply a sequence of data objects. A straight forward implementation of streams can be done using lists in Lisp. With a single fragment of a semantic net (a query pattern), the matching process runs through the copy of database entries one by one. This copy of the database entry list is given as an input stream (stream A in figure 14) for the matching process. Each entry is a message of the 'MAIL' database and contains pointers to several nodes containing its property values. For each database entry, the process which attempts the matching generates either a symbol indicating that the match has failed or the entry itself. The results for all of the given database entries are collected into a stream, which is passed through a filter to weed out the failures. The result is a stream of all the database entries that contain items matching the query pattern (see stream B in Figure 14).

In the QIRM system, a query takes a copy of all the database entries as an input stream for a 'where-processor' and performs the network-fragment matching operation for every entry in the stream as indicated in Figure 15. That is, for the given input stream (stream I), the where-processor generates a filtered new stream (stream II) consisting of all entries which have items satisfying the query condition. This filtered stream (II) is taken by a 'select-processor' to generate the final output of the query.

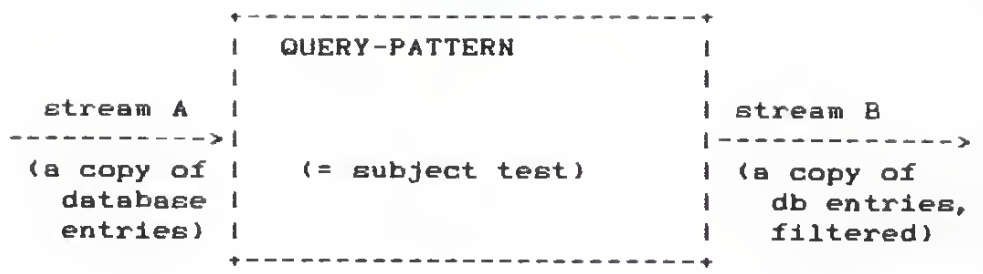

Figure 14. A query-pattern processes a stream of entries.

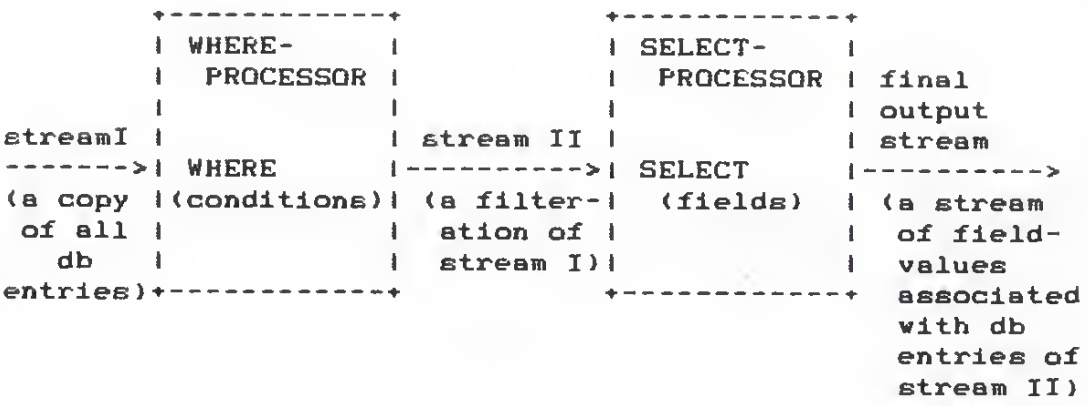

Figure 15. A query processor consists of a where-processor and a select-processor.

3.3.4.3. Simple Query and Stream Implementation

To answer a simple query, the system uses the query with an input stream consisting of copies of all database entries. The output stream from the where-processor contains the filtered entries. This stream of filtered database entries is then used to generate a stream of copies of the SELECTed field-values, and this is the stream that is finally printed at the terminal.

3.3.4.4. Compound Queries and Stream Implementation

The real elegance of the stream implementation is evident when compound queries are considered. The processing of compound queries makes use of the ability of the filter to demand that a match be consistent with a specified network fragment.

For example, to handle the 'and' of two compound-query-patterns, such as

```
( and  ( =  body  (rules and inheritance)  )
       ( =  to  faculty  )  )
```

(This query can be informally stated as: 'Find all messages whose text bodies contain the phrase "rules and inheritance" and whose receivers are faculty members.'.), the query processor first finds all entries containing the fragment that matches the following pattern: (= body (rules and inheritance)). This produces a stream of entries, each of whose body contains the phrase 'rules and inheritance'.

Having the new filtered stream, all entries that contain the fragment matching the following are found among the entries in the new stream: (= to faculty). The 'and' of two compound-query-patterns (see section 3.3.3.2) can be viewed as a series combination of the two component compound-query-patterns, as shown in Figure 16. The entries that pass through the first compound-query-pattern are filtered, and further filtered by the second compound-query-pattern.

Figure 17 shows the analogous method for computing the 'or' of two compound-query-patterns as a parallel combination of the two component compound-query-patterns. The input stream of entries is filtered separately by each compound-query-pattern. The two resulting streams are then merged (for example, by appending the streams and eliminating the duplicated entries) to produce the output stream of processing the 'or' clause.

From the stream-of-entries viewpoint, the 'not' of some query-pattern acts as a filter that removes all entries having items specified in the query-pattern. For instance, given the clause

(not (= from mary))

the system attempts, with the given database entries, to produce the stream of entries consisting of network fragments that satisfy (= from mary). Then, the system removes from the input stream all entries for which such fragments exist. The result is a stream consisting of only those entries in which the binding for 'from' does not satisfy (= from mary).

For example, in processing the query

```
(and  ( =  text  (key finding) )
      (not (=  from  mary ) )  )
```

the first clause will generate a stream of entries each of whose body contains the phrase 'key finding'. Taking 'and' with the not clause will filter the stream by removing all entries in which the bindings for 'from' satisfy the restriction that the message-sender's id-name is 'mary'.

The queries containing '<', '>', '<=', or '>=' as the predicate operator can be implemented with a similar filter on entry streams. The system uses each entry in the stream to instantiate the property variable (referred to as property-name) in the query pattern and then applies the Lisp predicate. Then the system removes from the input stream all entries for which the predicate fails.

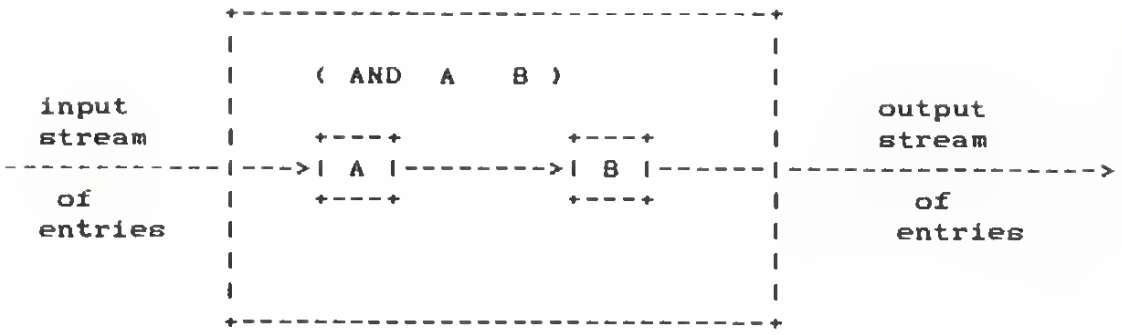

Figure 16. The 'and' combination of two compound-query-patterns

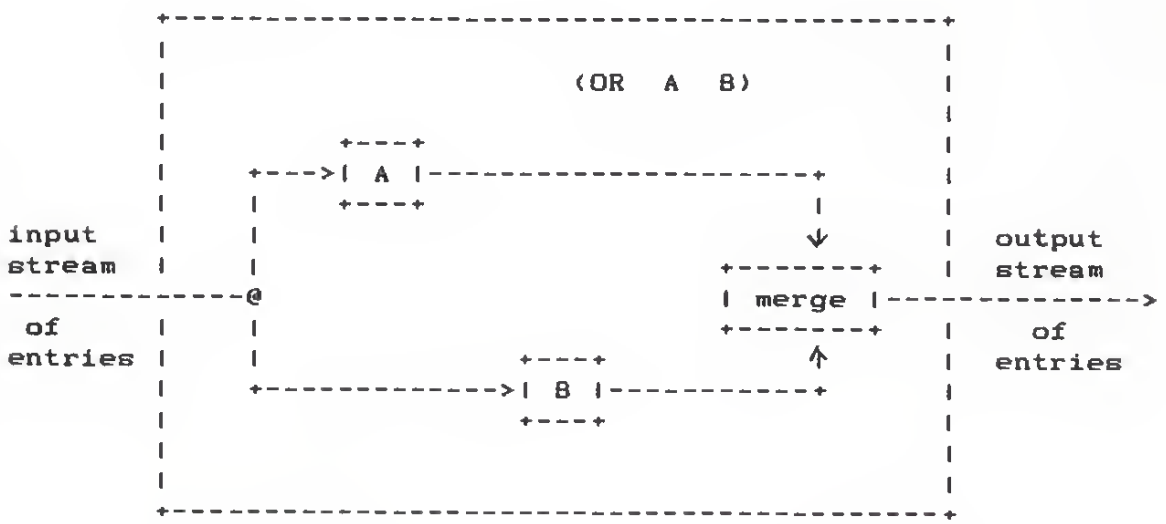

Figure 17. The 'or' combination of two compound-query-patterns

3.3.4.5. The Query Evaluator and the Query Driver

The function that coordinates the matching operations is called 'qeval', and it plays a role analogous to that of the 'eval' function for Lisp. 'qeval' takes as inputs a query and a stream of a copy of the database entries. Its output is a stream of selected field-values of the database entries, corresponding to successful matches to the query pattern, as indicated in Figure 15. Like 'eval', 'qeval' classifies the different types of expressions (query-patterns) and dispatches each to an appropriate function. There is a function for each special form such as and, or, not, <, >, <=, >=, and =.

The driver loop reads a user request from the terminal which is specified following SELECT-WHERE prompts. The SELECT and the WHERE clause indicate the SELECTed property links and the condition of the messages in which a user wants to retrieve, respectively. For each query, it calls qeval with the WHERE clause and a stream that consists of a copy of all of the database entries. This will produce the stream of entries which are the result of all possible matches performed by a 'where-processor' (refer Figure 15 and section 3.3.4.2). For each entry in this resulting stream, it instantiates the SELECTed fields' values in the entry. This stream of instantiated fields' values is then printed with the associated message number.

3.4. SUMMARY

QIRM is a prototype information retrieval system that is designed to provide not only a functional enhancement to the 4.3 BSD Unix mail facility, but also some insight into the incorporation of Artificial Intelligence techniques to the Unix mail facility.

The database in QIRM is based on the semantic network structure and also utilizes an indexing scheme. By employing this approach, the entire database can be searched very easily. The process for loading the data into a database involves determining the data source, constructing a database from the data file, and storing the database in a file. The decision, concerning which file is used for the source of the data, is based upon the existence of an old mail database file. This approach distinguishes the first-time user from the regular user of the system and minimizes the latter's waiting time for loading a database.

The QIRM system increases the selectivity of information retrieved by allowing the user to specify a request using the mail-query language. The query language used in this system consists of the simple query and the compound query. This language provides a user-friendly interface and a pattern-directed access to the messages. The query processor is organized around the network-fragment matching and the stream implementation. A query takes an input stream of database entries and performs the matching operation for every entry in the stream. As its output, the query

generates a new stream consisting of all SELECTed field values associated with the messages which have structures satisfying the query's WHERE condition.

CHAPTER 4

CONCLUSIONS AND POSSIBLE ENHANCEMENTS

The characteristics of the approach which is based on structuring the mailbox by organizing the contents of messages, have been illustrated through the description of the QIRM prototype system. Also, a brief review of electronic mailbox systems, concepts, and tools used in the prototype has been provided.

The QIRM system provides a user friendly interface using a mail-query language based on the tuple calculus, and employes techniques from artificial intelligence. Users can query messages by specifying the category of the message they want to retreive. This approach provides an information organization and dramatically increases the selectivity of information retrieved.

Based upon the concepts and features of QIRM, it has been demonstrated that the work on information retrieval systems and database management systems is potentially relevant to the design of structured electronic mail systems.

It is desirable that QIRM would be built directly on the top of the existing Unix mail facilities. In this way, users could continue to operate on their mail as usual, and also access the messages in the more flexible ways which QIRM provides. Such a system could provide facilities for updating each user's database file automatically, whenever a

message has arrived in or has been deleted from the mailbox. With this automatic updating, the query system could maximize both efficiency and the user friendliness by eliminating the database-loading process which currently is needed before the retrieval facilities of QIRM can be utilized.

Many facilities could be added to enhance the intelligence of the system. For example, in order to perform more intelligent interpretation of the messages, the system could also allow user-specified rules which automatically screen messages arriving in a user's mailbox. Also, this mechanism could be used to sort messages into different categories according to the individual user's preference. These sorting facilities could examine the fields and body presented in the mail and deduce the message classification based on its set of rules. In addition, the system could employ these rules to present the user with an overview of the messages currently available; thus, a user could easily pick what is of interest.

In the current Unix mail system, the bare login name is used as a user-id, only when a message is sent to the person on the same machine. For example, if one wants to send messages to people on the Arpanet, a recipient's id has the form "id@host". 'Id' is the login name of the recipient and 'host' is the name of the machine where the recipient can be found on Arpanet. The way of specifying the user's id varies in a manner which depends on the type of network involved. This mechanism raises one drawback of the QIRM system: Since QIRM recognizes a bare login name and a login

name which is a concatenation of the bare login name and the strings (e.g. the names of systems) as different user-id, querying messages based on the user-id sometimes produces unexpected output. Another drawback is that QIRM does not provide an intelligent function of determining the identity of an object (the partial property value). For example 'farmer', 'farmer!', and 'farmers' are recognized as different objects whereas all three should be interpreted as identical. Such a problem is caused by this system's search procedure. In order to perform fast searching, this procedure scans and recognizes an object not character by character but word by word. (A word indicates a sequence of any characters except blank.) Since it is more desirable to develop a system which is both efficient and intelligent, the way for accomplishing such a result should be sought.

BIBLIOGRAPHIES

[1]. P. A. Wilson, 'Structures for Mailbox System Applications', Proceedings of the IFIP 6.5 Working Conference on Computer-Based Message Services, May 1984, North Holland Publications, 1984, pp 149 - 165.

[2]. Jacob Palme, 'You Have 134 Unread Mail! Do You Want To Read Them Now?', Proceedings of the IFIP 6.5 Working Conference on Computer-Based Message Services, May 1984, North Holland Publications, 1984, pp 175 - 184.

[3]. Robert Wilenskey, 'Lisp Craft', W.W. Norton & Company, 1984.

[4]. Thomas W. Malone, Kenneth R. Grant, Franklyn A. Turbak, Stephen A. Brobst, and Michael D. Cohen, 'Intelligent Information-Sharing Systems', Communications of ACM, Vol. 30, No. 5, May 1987, pp 390 - 402.

[5]. Dennis Tsichritzis, Fausto A. Rabitti, Simon Gibbs, Oscar Nierstrasz, and John Hogg, 'A System for Managing Structured Messages', IEEE Transactions on Communications, Vol. com-30, No. 1, Jan. 1982, pp 66 - 73.

[6]. P. A. Wilson, 'Applications and Structures for Mailbox System', Data Communications: The Wired Society, P. D. English, Maidenhead Berks., England: Pergamon Infotech., 1983, pp 73 - 94.

[7]. Matthias Jarke, Jurgen Koch, and Joachim W. Schmidt, 'Introduction to Query Processing', Query Processing in Database Systems, Springer-Verlag Berlin Heidelberg, 1985, pp 3 - 28.

[8]. Gordon McCalla and Nick Cercone, 'Approaches to Knowledge Representation', Computer, IEEE Computer Society, Vol 16, No. 10, Oct 1983, pp 12 - 18.

[9]. Ronald J. Brachman, 'What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks', Computer, IEEE Computer Society, Vol. 16, No. 10, Oct 1983, pp 30 - 36.

[10]. Kurt Shoens and Craig Leres, 'Mail Reference Manual Version 5.2', Apr. 1986.

[11]. Julian Newman, 'Contracts Made by Electronic Mail: Legal Issues, Technology, and Services', Proceedings of the IFIP 6.5 Working Conference on Computer-Based Message Services, May 1984, North Holland Publications, 1984, pp 237 - 246.

[12]. Chamberlin, D. D. et. al., 'SEQUEL2: a unified approach to data definition, manipulation and control', IBM Journal Research and Development, Vol. 20, No. 6, Nov. 1976, pp 560 - 575.

[13]. S. M. Deen, 'Principles and Practice of Database Systems', MacMillan Publishers Ltd, 1985.

[14]. Alan Mayne and Michael B Wood, 'Introducing Relational Database', NCC Publications, 1983.

[15]. William D. Haseman and A. B. Whinston, 'Introduction to Data Management', Richard D. Irwin Inc., 1977.

[16]. John K. Foderaro, K. L. Sklower, and Kevin Layer, 'The FRANZ LISP Manual', 1983.

[17]. Elaine Rich, 'Artificial Intelligence', McGrow-Hill Book Company, 1983.

[18]. J. A. Welch and P. A. Wilson, 'Electronic Mail Systems - A Practical Evaluation Guide', NCC Publications, 1981.

[19]. Jeffrey D. Ullman, 'Principles of Database systems', Computer Science Press, 1982.

[20]. Peter Vervest, 'Electronic Mail and Message Handling', Quorum Books, 1985.

[21]. Avron Barr and E. A. Feigenbaum, 'The Handbook of Artificial Intelligence Volume I & II', HeurisTech Press, 1981.

[22]. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, 'The Design and Analysis of Computer Algorithms', Addison-Wesley Publishing Company, 1974.

APPENDIX A

ON-LINE MANUAL

This appendix contains an on-line manual for the users. It describes the user-interface of QIRM, and the syntax of the mail-query language used by users for specifying their request on retrieving messages.

NAME

QIRM - retrieve mail

SYNOPSIS

QIRM

DESCRIPTION

QIRM is a Query system for Information Retrieval in a Mailbox (QIRM).

QIRM reads messages in a user's mailbox (or messages in both a mailbox and an old mail database file) and constructs a database. An old mail database file arrives at by concatenating a user's login-id and a string 'data'. For example, if 'mary' is the user's login-id, then old mail database file is named 'marydata'. The database in 'marydata' is the one that had been constructed when QIRM was called the last time by a user, 'mary'.

After the database is constructed, QIRM asks the user to categorize the messages which he/she wants to retrieve. The user can specify the fields and conditions of such messages by posing queries in response to the system's SELECT-WHERE prompts. The system's response to the user's query involves the number of messages retrieved and the first item of the messages, if there are more than one message recalled. In such a case, QIRM gives the number of additional response records and interrogates the user about their disposition. That is, the user can either continue accessing the remaining messages by choosing 'y' as the answer to the system's 'More?(y/n)' question or start a new dialogue with the new SELECT-WHERE prompts. The user can exit the system by SELECTing 'stop' command.

** Simple Queries **

The syntax of simple query is

SELECT (<field>+ | *)
WHERE (<query-pattern>) | ()

Some points concerning this query are:

1. a. <field>+ indicates the set of length one or more which consists of elements of the form <field>
where, in turn, is made up of sdate, rdate, form, to, subject, status, and body indicating the date of sending a message, the date of receiving a message, the message-sender's login-id, the message-recipient's login-id, the subject of a message, the status of a message, and the body of a message, respectively.
   b. '|' indicates alteration.
   c. '*' is the shorthand of all fields.

2.
2.1. <query-pattern> has the following structure.

<predicate-operator> <property> <propertyvalue>

where
a. <predicate-operator> can be <. >, <=, >=, =.
b. <property> can consist of sw, sd, sm, sy, rw, rd, rm, ry, from, to, subject, status, and text.
'sw, sd, sm, sy' indicate the week day, the day number, the month, the year of sending a message, respectively. 'rw, rd, rm, ry' indicate the week day, the day number, the month, the year of sending a message, respectively. 'text' indicates a phrase or words in a message body. (A word is a sequence of any characters except blank.)

   c. 1. <propertyvalue> can be the value to be searhed for in the named property of messages. In the case of the value for 'text' property, either the exact subject of a message or a word out of the message-subject can be specified.

      2. If the property value consists of more than one word, it should be parenthesized, otherwise, it should not.

2.2. () indicates that 'no condition' is specified for retrieving messages.

3. QIRM does not distinguish between upper and lower case letters.

** Compound Queries **

The syntax of a compound query is

```
SELECT  ( <field>+  )  |   (*)
WHERE   <ANDOR-query> | <NOT-query>
```

Some points concerning this query are:

1. <ANDOR-query> has the following form:

   (and | or <compound-query-pattern>++)

   where
   a. <compound-query-pattern>++ indicates the set consisting of two or more <compound-query-pattern>s.

   b. <compound-query-pattern> can be any of the following:

   ```
   (and  <compound-query-pattern>++  )
   (or  <compound-query-pattern)++  )
   (not ( <query-pattern> ) )
   (  <query-pattern>  )
   ```

2. <Not-query> has the following form:

   (not (<query-pattern>))

APPENDIX B

SOURCE CODE LISTING

This appendix consists of two listings of the source code for implementing QIRM. QIRM.l contains a main source listing in Lisp, and ml.c comprises a source listing of 'C' function which is used to obtain the name of a user's mailbox.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  This system function causes the Lisp system to ~~
~~  go through the transfer tables and reset all    ~~
~~  the appropriate links.                           ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(sstatus translink on)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  Declare special variables for the compiler.  ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(declare (special inport dataport outport st number MAIL ))
(declare (special dummyport idname chn change done))

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  In order to make 'reader' case-insensitive, the       ~~
~~  reader is modified to conform with UCI-Lisp syntax.   ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(cvttoucillsp)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  Set the syntax class of these characters to           ~~
~~  syntaxclass in the current readtable.                 ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(setsyntax '/! 'vcharacter)
(setsyntax '/. 'vcharacter)
(setsyntax '/' 'vcharacter)
(setsyntax '/" 'vcharacter)
(setsyntax '/[ 'vcharacter)
(setsyntax '/] 'vcharacter)
(setsyntax '// 'vcharacter)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  MAIN FUNCTION OF the QIRM SYSTEM                      ~~
~~                                                        ~~
~~  This function begins with calling a function          ~~
~~  'get-mailboxfile' to access a user's mailbox and check ~~
~~  if the mailbox is empty.   If it's empty, a proper    ~~
~~  message is printed at the terminal and a system is    ~~
~~  terminated.  Otherwise, this function calls 'setup    ~~
~~  -db' which determines the source of input data and    ~~
~~  loads the input data into a database.                 ~~
~~  If the system finds any difference between the mail   ~~
~~  in the current mailbox and the mail in 'old-db', it   ~~
~~  calls 'save-db-in-file' to update 'old-db'.  'Old-db' ~~
~~  is an old database file containing the database which ~~
~~  was constructed form the most previous mailbox.       ~~
~~  Finally, 'query-driver' is called in order for        ~~
```

```
                (t  (go more))))
            ( (= ch -1) (return t) )
            (t (readc inport) (go more))))) )


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  This function saves the database in a file 'old-db'              ~~
~~  in the list format of Lisp.   The file is named                  ~~
~~  by concatenating the user's login-id and the string              ~~
~~  'data'.                                                          ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun save-do-in-file()
  (declare (special outport))
  (prog(n l mname)
     (setq outport (outfile (concat idname 'data)))
     (setq n 1)
     (setq l (length MAIL))
       loop
     (setq mname (concat 'msg n))
     (princ '/( outport)
     (wrt-db (get mname 'rdate))
     (wrt-db (get mname 'sdate))
     (wrt-db (get mname 'to))
     (wrt-db (get mname 'from))
     (wrt-db (get mname 'pos-linenum))
     (wrt-db (get mname 'text))
     (wrt-db (get mname 'status))
     (wrt-db (get mname 'subject))
     (princ '/) outport)
     (setq n (1+ n))
     (cond ((> n l)  (close outport) )
           (t (go loop)))))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  This function uses 'cfasl'  to load a foreign                    ~~
~~  function 'ml.c' (written in 'C') into the lisp                  ~~
~~  system.    Using the user-id returned from 'ml.c',              ~~
~~  this function obtains the user's mailbox-id in the              ~~
~~  proper form through a subfunction 'id-string'.                  ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun get-mailboxfile()
  (declare (special id))
  ( cfasl 'ml.o '_ml 'ml "integer-function")
  (setq  id (new-vectori-byte 20))
  (setq idname (id-string (ml id) "" id 0) )
  (setq inport (infile
                (concat '//usr//spool//mail//  idname))))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
~~   A function to transform the value returned from     ~~
~~   'ml.c' to a desired format (a string).              ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun id-string( num  name io i)
  (cond
       ((= 0  num) name)
       (t (id-string (1- num)
            (concat  name (ascii (vrefi-byte id i)))
            io  (1+ i)))))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~   This function, first, checks the existance of       ~~
~~   'old-db'.  If 'old-db' found, it reads a msg        ~~
~~  in 'old-db' comparing the key portion of the same    ~~
~~   msg of mailbox ('line') with that of 'old-db'.      ~~
~~   If two messages compared are identical, contents    ~~
~~   of the message are added to the database 'MAIL'     ~~
~~   and the result of the comparison is returned.       ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun compare(line)
  (prog(data p msgname)
  (setq msgname (concat 'msg number))
    loop
  (setq data (read dataport))
  (cond ( (null data) (return 'end))
        ( (equal line (car data))
          (setq p (read-one-msg))
          (putprop msgname line 'rdate)
          (putprop msgname (nth 1 data) 'sdate)
          (putprop msgname (nth 2 data) 'to)
          (putprop msgname (nth 3 data) 'from)
          (putprop msgname (list p  (cadr (nth 4 data)))
                                'pos-linenum)
          (putprop msgname (nth 5 data) 'text)
          (putprop msgname st 'status)
          (putprop msgname (nth 7 data) 'subject)
          (setq number (1+ number))
          (setq MAIL (cons msgname MAIL))
          (return 'ok))
        (t (setq change 1) (go loop)))))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~   A function to read single msg (header + body) in    ~~
~~   'old-db'.                                           ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun read-one-msg()
  (prog(line)
    again
  (setq line (lineread inport t))
```

```
  (cond ( (null line)
          (return (filepos inport)) )
        (t (if (eq (car line) 'status:)
               (setq st line)) (go again)))))


 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ~~  A subfunction of 'save-db-in-file' to write                      ~~
 ~~  the database in 'old-db'.                                        ~~
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defun wrt-db(data)
  (prog(listdata)
    (if (listp (car data))
     then (patom "(" outport) (setq listdata 1) )
  keep-writing
    (cond ( (null data)
            (if (= listdata 1) (patom ")" outport) )    )
          ( (atom (car data))
            (patom data outport) (terpri outport) )
          ( t  (patom (car data) outport) (terpri outport)
               (setq data (cdr data)) (go keep-writing)))))


 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ~~  The function builds a database by reading mailbox as            ~~
 ~~  an input file.   After the contents of each message             ~~
 ~~  are constructed into the database, the message name             ~~
 ~~  is added to a glooal variable 'MAIL'.                           ~~
 ~~  'Line' is the first line of a msg which was read                ~~
 ~~  from 'mailbox',  'num-line' is the number of lines              ~~
 ~~  of the current header field,                                    ~~
 ~~  'headerp' is a boolean variable to show the current             ~~
 ~~  line is in the header part or in the body part                  ~~
 ~~  (1: header  0: body),  'mailnum' is a integer                   ~~
 ~~  indicating the msg number.                                      ~~
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defun build-db( line num-line headerp mailnum)
  (setq change 1)
  (prog(a  name position )
   (setq name (concat 'msg mailnum))
    loop
   (cond ( (= headerp 1)
           (setq a (lineread inport t))
           (cond ( (not(null a))
                   (cond ( (eq '/:
                               (car (last (explode (car a)))))
                          (cond ((= num-line 1)
                                 (add-db (car line) name
                                         line))
                                (t (add-db (caar line)
                                           name line)
                                   (setq num-line 1) ) )
```

```
                        (setq line a)
                        (go loop))
                       (t
                        (cond ((= num-line 1)
                               (setq line
                                (appendl ((ist line)  a)))
                              (t (setq (ine
                                  (appendl (line  a))) )
                        (setq num-line (1+  num-line))
                                    (go loop) ) ))
              (t (setq headerp 0)
                 (setq position (filepos inport))
                 (cond ((= num-line 1)
                        (add-db (car line) name (ine))
                        (t (add-db (caar line) name line)
                           (setq num-line 1)))
                           (go loop) )) )
        (t (add-db 'Pos-linenum  name
                   (list position (msg-body mailnum)))
           (setq MAIL  (cons name MAIL))
           (cond ((= done 0)
                  (setq line (lineread inport))
                  (setq headerp 1)
                  (setq mailnum (1+  mailnum))
                  (setq name (concat 'msg mailnum))
                  (go loop))
                  )) ) ))
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~   A subfunction called by 'build-db' to add a line of   ~~
~~   data to the database.                                  ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defun add-db (field mailname lst)
  (caseq field
         (From  (putprop mailname (cdr lst) 'rdate))
         (Date: (putprop mailname lst 'sdate))
         (From: (putprop mailname lst 'from))
         (To:   (putprop mailname lst 'to))
         (Status: (putprop mailname lst 'status))
         (Subject: (putprop mailname (st 'subject))
         (Pos-linenum (putprop mailname lst 'pos-linenum))
         (Text (putprop mailname (cdr lst) 'text))
         (Apparently-To: (putprop mailname (st 'to)) ))
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~   'Msg-body' is a function to read the body of a msg    ~~
~~   and add this data to the database.                    ~~
~~   'R-one' is a sub-function to read one line of a msg   ~~
~~   body and return it.                                   ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defun msg-body (mailnum )
  (prog(oneline text char num-textline)
  (setq num-textline 0)
  (setq text '("""))
   loop
  (setq char (tyipeek inport)) (setq oneline (r-one))
  (cond ( (= done 1)
          (add-db 'Text (concat 'msg mailnum) text)
          (return  num-textline)  )
        ( (and (= char 70) (eq (car oneline) 'From  ) )
          (add-db 'From (concat 'msg (1+ mailnum))  oneline)
          (add-db 'Text (concat 'msg mailnum) text)
          (return  num-textline ))
        (t (if  (not(null oneline))
                (setq text (append1 text oneline )))
           (setq num-textline (1+ num-textline))
           (go loop) ) )))



(defun r-one()
  (prog(c ch)
   lop
  (setq  ch (tyipeek inport) )
  (cond ((= ch 32) (readc inport) (go lop) )
        ((> ch 32) (if (memq ch '(40 41 ))
                       then (readc inport)
                       else
                       (setq c (append1 c (ratom inport))))
                       (go lop) )
        ((= ch 10) (readc inport) (return c))
        ((= ch -1) (setq done 1) (return c))
        (t  (readc inport) (go lop)) )))
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  This main function of query system scans a user-                  ~~
~~  query, passes it to the query evaluator 'qeval'                   ~~
~~  together with a copy of a stream of entries of the                ~~
~~  database.                                                         ~~
~~  The result of the evaluation is a stream of selected              ~~
~~  field-values of database entries which contain items              ~~
~~  satisfying the query.          The final stream is               ~~
~~  printed at the terminal by 'prt-output', 'prt-mail'              ~~
~~  'printtext1', 'printtext2', 'prtlist'.                           ~~
~~  'Prt-output' and 'prtmail' is the functions to print              ~~
~~  output values of user-selected fields at the                     ~~
~~  terminal.                                                         ~~
~~  'Printtext1' and 'printtext2' are the functions for               ~~
~~  printing a message body: the former is for the text               ~~
~~  body with more than 22 lines and the latter is for               ~~
~~  the body with equal or less than 22 lines of text.               ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
(defun query-driver()
  (prog(fd field q)
    more-query
   (terpri)
   (princ "SELECT  ")
   (setq field (read) )
   (if (and (listp field) (eq '* (car field))
            (null (cdr field)))
    then
    (setq fd '(rdate sdate to from suoject status body))
    else (setq fd (contents field (type-check  field))))
  (cond  ( (eq fd 'stop) (princ  "***  bye bye  **")
                         (return '*) )
         ( (eq fd 'error)
           (go more-query) )
         (t (princ "WHERE  ")
            (setq q (read))
            (if (null q) then (prt-output fd MAIL)
             else (prt-output fd (qeval q MAIL ) ) )
            ( go more-query) ))))


(defun prt-output( field lst )
  (cond ( (eq 'error lst)
          (princ "** ERROk - Unknown expression **"))
        (t (if (= 0 (length lst))
            then (msg "****  No message Recalled!  ****")
                 (terpri) (terpri)
            else
             (msg "****  " (length lst)
                  " message Recalled! ****" )
              (terpri) (terpri)
              (msg  "& "  (car lst)) (terpri)
              (prtmail  field lst) ))) )


(defun prtmail(field lst)
  (prog(fld m-num output field2 linenum)
   (setq field2 field)
    keep-printing
   (setq fld (car field2) )
   (setq m-num (car lst) )
   (cond  ( (eq fld 'body)
            (if (null (get m-num 'text))
             then (princ  "** no message body found  **" )
                  (terpri)
             else
             (setq linenum (nth 1 (get m-num 'pos-linenum)))
             (if (> linenum 22)
              then
              (printtext1 (nth 0 (get m-num 'pos-linenum))
                          linenum)
               else
               (printtext2 (nth 0 (get m-num 'pos-linenum))
```

```
                              linenum))))
              ( (eq fld 'sdate) (princ "sdate: ")
                (princ (cdr (get m-num fld))) (terpri) )
              ( (eq flo 'rdate) (princ "rdate: ")
                (princ (cdr (get m-num fld))) (terpri) )
              (t  (setq output (get m-num fld))
                  (cond ((null output)
                         (msg "**  no " fld " found  **"))
                         (t
                           (if (atom (car output))
                               (princ output)
                               (prtlist output) )))
                  (terpri)  ) )
    (setq field2 (cdr field2) )
    (if  (null field2)
     then (terpri)
          (if  (neq 1 (length lst))
           then (msg (1- (length lst))
                     " message left!  More?(y//n)   ")
                (if (yes) then
                    (terpri)
                    (msg "& "  (cadr lst))
                 else  (setq lst '() )   ))
          (terpri)
          (setq lst (cdr lst))
          (setq field2 field) )
    (if (null lst) (terpri) (go keep-printing))   ))


(defun printtext2(pos linenum)
  (fseek inport pos 0)
  (terpri)
  (prog(a)
   loop
   (setq a (readc inport) )
   (princ a)
   (cond [ (eq  a (ascii 10) )
           (setq linenum (sub1 linenum))
           (if (= 0 linenum) (return t) (go loop)) ]
         [t (go loop)] )))


(defun printtext1(pos linenum)
  (fseek inport pos 0)
  (terpri)
  (prog(a  n)
   (setq n 0)
    loop
   (setq a (readc inport) )
   (princ a)
   (cond [ (eq  a (ascii 10) ) (setq n  (add1 n))
                               (setq linenum (sub1 linenum))
           (if (> linenum 0)
            then (if (>= n 20)
                 then (princ " MORE")
                      (if (eq (tyi) 10) (go loop)
```

```
                        (return t))
                    else (go loop))
               else  (= 0 linenum) (return t) (go loop)) ]
          [t (go loop)] )))


(defun prtlist(lst)
  (cond  ( (not (null lst)) (princ (car lst)) (terpri)
           (prtlist (cdr lst)) )) )



~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  'Yes', 'type-check', and 'contents' are utility      ~~
~~  functions.  'Yes' returns 'true' if a user types     ~~
~~  'y'.  Otherwise, this function returns 'nil'.        ~~
~~  'Contents' and 'type-check' are functions called by  ~~
~~  'query-driver' to check the syntax of a user query.  ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun yes()
  (if (equal (read)  'y) t  nil) )


(defun type-check(exp)
  (cond ( (eq exp 'stop) 'stop)
        ( (or (atom exp) (null exp))
        (msg  "** Unknown expression type -- " exp  " **")
              'error )
        (t exp ) ))



(defun contents(exp1 exp2)
  (cond ( (eq  'error exp2) 'error)
        ( (eq 'stop exp2) 'stop)
        ( (null exp2) exp1 )
        ( (memq (car exp2) (q-field-list) )
          (contents exp1 (cdr exp2)) )
        (t (msg  "** Unknown field -- " exp2 " **")
           'error) ))



~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  'Qeval' takes a query and a stream of a copy of      ~~
~~  database entries as inputs.  It classifies the      ~~
~~  different types of expressions ano dispatches to    ~~
~~  an appropriate function for each.                    ~~
~~  'Check-domain' is calleo for integrity-checking     ~~
~~  of property-values of user-queries.                  ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun qeval(q frame)
  (if (atom q) 'error
     (caseq (car q)
```

```
        (and (if (null (cddr q)) then 'error
              else (conjoin  (cdr q)  frame)))
        (or  (if (null (cddr q))  then 'error
              else (disjoin  (cdr q) frame)))
        (not (negate  (cdr q) frame))
        (=   (equate (nth 0 (cdr q)) (nth 1 (cdr q)) frame))
        (<=  (less-eq (check-domain (cor q)
                                    (cadr q) (caddr q)) frame))
        (>=  (greater-eq (check-domain (cdr q)
                                    (cadr q) (caddr q)) frame))
        (<   (less (check-domain (cdr q) (cadr q)(caddr q))
                                    (cadr q) (caddr q) frame))
        (>   (greater (check-domain (cdr q)
                                    (cadr q)(caddr q))
                                    (cadr q) (caddr q) frame))
        (t   'error ) )) )



(defun check-domain(q property value)
  ( cond ( (memq property '(rm sm))
           (if (memq value (monthlist)) q 'error))
         ( (memq property '(rd sd ry sy))
           (if (numberp value) q 'error) )
         ( (memq property '(rw sw))
           (if (memq value (weeklist)) q 'error))  ))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  This function takes a property-name and an entry          ~~
~~  (a message), and returns the value of given               ~~
~~  property of the message.                                  ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


(defun get-property-value(property l)
  (caseq  property
    (rw   (nth 1 (get l 'rdate))  )
    (rm   (nth 2 (get l 'rdate)) )
    (rd   (nth 3 (get l 'rdate)) )
    (ry   (get_pname (implode (cddr
            (explode(nth 5 (get l 'rdate))))))  )
    (sw   (nth 1 (get l 'sdate)) )
    (sm   (nth 3 (get l 'sdate)) )
    (sd   (nth 2 (get l 'sdate)) )
    (sy   (ntn 4 (get l 'sdate)) )
    (t    (cdr (get l property) ))  ))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  'Equate' handles the query with '=' as its predicate     ~~
~~  operator.                                                ~~
~~  If the given property is 'text',  then it               ~~
~~  calls 'atom-text-equate' or 'list-text-equate'           ~~
```

```
~~  according to the type of property-value which a user ~~
~~  specified in his query.  That is, if the property-   ~~
~~  value consists of more than one word, 'list-text-    ~~
~~  equate' is called.  Otherwise, 'atom-text-equate' is ~~
~~  called.  A word indicates a sequence of any          ~~
~~  characters except blank.                             ~~
~~  If the given property is 'subject', 'subject-equate' ~~
~~  is called.                                           ~~
~~  For the queries excluding those with 'text' or       ~~
~~  'subject' as their properties, 'other-equate' is     ~~
~~  called to handle a query with '='.                   ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun equate(property qvalue l)
  (cond ( (eq property 'text)
          (if (atom qvalue)
              (atom-text-equate qvalue l)
              (list-text-equate qvalue l)) )
        ( (eq property 'subject)
          (subject-equate property qvalue l) )
        ( (memq property
          '(to from subject status rw rd sw sd rm sm ry sy))
          (other-equate property qvalue l))
        (t 'error)))


(defun other-equate(pty qvalue l)
  (prog (frame prop-value)
     (cond ( (eq pty 'sw) (setq qvalue (concat qvalue ",")))
           ( (eq pty 'ry)
             (setq qvalue (get_pname (concat qvalue "" )))))
    loop
     (setq prop-value (get-property-value pty (car l)))
     (cond ( (and (atom qvalue) (listp prop-value)
                  (or (memq ovalue prop-value)
                      (memq (concat qvalue ",")
                            prop-value)) )
             (setq frame (cons (car l) frame)))
           ( (and (atom prop-value)
             (equal qvalue prop-value))
             (setq frame (cons (car l) frame)) ))
     (setq l (cdr l))
     (if (null l) (return frame) (go loop) ) ))



(defun subject-equate(pty qvalue l)
  (prog (frame prop-value )
    loop
   (setq prop-value (get-property-value pty (car l)))
   (cond ( (and (atom qvalue)
                (or (memq ovalue prop-value)
                    (memq (concat qvalue ",") prop-value)
                    (memq (concat qvalue ".") prop-value)) )
           (setq frame (cons (car l) frame)))
         ( (and (listp qvalue) (equal qvalue prop-value))
```

```
            (setq frame (cons (car l) frame)) ) )
    (setq l (cdr l))
    (if (null l) (return frame) (go loop) ) ))


(defun atom-text-equate(qvalue lst)
  (prog(frame prop-value found)
    loop1
    (setq prop-value (get (car lst) 'text) )
    (prog(onelist)
     loop2
     (cond ( (null prop-value) (setq founo 0) )
           ( (listp (car prop-value))
             (setq onelist (car prop-value))
             (if (or (memq qvalue onelist)
                     (memq (concat qvalue '/,) onelist)
                     (memq (concat qvalue '/.) onelist) )
              then (setq found 1)
              else (setq prop-value (cdr prop-value))
                   (go loop2)) )
           ( t
             (if (or (eq (car prop-value) qvalue)
                     (eq (car prop-value)
                         (concat qvalue '/,))
                     (eq (car prop-value)
                          (concat qvalue '/.)) )
              then (setq found 1)
              else (setq prop-value (cdr prop-value))
                   (go loop2))) ))
     (if (= found 1)
      then (setq frame (cons (car lst) frame)))
     (setq lst (cdr lst) )
     (if (null lst) (return frame) (go loop1))  ))


(defun list-text-equate(qvalue lst)
  (prog(frame prop-value  found)
   loool
   (setq found 0)
   (setq prop-value (get (car lst) 'text) )
   (orog(q qatom old-prop onelist)
     lsetq q qvalue)
     loop2
     (setq qatom (car q))
     (cond ( (null prop-value) (setq found 0) )
           ( (listp (car prop-value) )
             (setq onelist (car prop-value))
           (if (or (memq qatom onelist)
                   (memq (concat qatom '/,) onelist)
                   (memq (concat qatom '/.) onelist) )
            then (setq found
                     (confirm qvalue onelist prop-value 0))
            else (setq proo-value (cdr prop-value))
                 (go loop2)) )
```

```
            ( t
               (if (or (eq (car prop-value) qatom)
                       (eq (car prop-value)
                           (concat qatom '/,))
                       (eq (car prop-value)
                           (concat qatom '/.)) )
                 then (if (= found 0)
                       then (setq found 1)
                            (setq old-prop prop-value) )
                       (setq q (cdr q))
                       (setq prop-value (cdr prop-value))
                       (if (null q) (setq found 2)
                           (go loop2))
                 else  (if (= found 1)
                        then (setq prop-value (cdr old-prop))
                             (setq q qvalue)
                             (setq found 0)
                        else
                        (setq prop-value (cdr prop-value)) )
                        (go loop2) ) )))
     (if (= found 2) then
         (setq frame (cons (car lst) frame)))
     (setq lst (cdr lst) )
     (if (null lst) (return frame) (go loop1))  ))


(defun confirm(qvalue onelist p-value f)
  (prog(word qatom oldp-value sameq)
    (setq sameq qvalue)
    loop
    (setq qatom (car sameq) )
    (cond ( (null p-value) (return 0) )
          (t
            (setq word (car onelist))
            (if (or (eq word qatom)
                    (eq word (concat qatom  '/,))
                    (eq word (concat qatom  '/.)) )
             then (if (= f 0)
                   then
                   (setq oldp-value p-value) (setq f 1) )
                  (setq sameq (cdr sameq))
                  (setq onelist (cdr onelist))
                  (if (null onelist)
                   then (setq p-value (cdr p-value))
                        (setq onelist (car p-value)) )
                  (if (null sameo) (return 2) (go loop) )
             else (if (= f 1)
                   then (setq p-value (cdr oldp-value) )
                        (setq onellst (car p-value))
                        (if (null onelist)
                         then (setq p-value (cdr p-value))
                              (setq onelist (car p-value)))
                         (setq sameq qvalue)
                         (setq f 0)
                    else (setq  onelist (cdr onelist))
```

```
                    (if (null onelist)
                     then (setq p-value (cdr p-value))
                          (setq onelist (car p-value))))
                (go loop) ) ))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  'Or' queries are handled by 'disjoin' and                              ;;
;;  'mergel'.   The output streams for the various                         ;;
;;  disjuncts of the 'or' are computed separately                          ;;
;;  and then merged.                                                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun disjoin(q-list frame)
  (cond ((null q-list)  '())
        (t (mergel (qeval (car q-list) frame)
           (disjoin (cdr q-list) frame)) )))


(defun mergel(lst1 lst2)
  (cond ( (or (eq lst1 'error) (eq lst2 'error)) 'error)
          ((null lst2) lst1 )
        ((memq (car lst2) lst1) (mergel lst1 (cdr lst2) ))
        (t (cons (car lst2)(mergel lst1 (cdr lst2))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  'And' queries are  handled by this function.                           ;;
;;  'Conjoin' takes as inputs the conjuncts and a                          ;;
;;  stream of entries, and returns the stream of fil-                      ;;
;;  tered entries.                                                         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun conjoin(q-list entries)
  (cond ((eq entries 'error) 'error)
        ((null q-list) entries)
        (t (conjoin (cdr q-list)
           (qeval (car q-list) entries)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  'Not' queries are handled by 'negate' function.                        ;;
;;  Given a query and a stream of entries, it returns                      ;;
;;  a stream of entries which don't contain property-                      ;;
;;  values satisfying the queries.                                         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun negate(q frame)
    (diff   frame   (qeval (car q)  frame)))
```

```
(defun diff(lst1 lst2)
  (cond ( (eq lst2 'error) 'error )
        ( (null lst2) lst1 )
        (t (diff (del (car lst2) lst1)  (cdr lst2))) ))

(oefun del(elem lst)
  (cond ( (null lst) '() )
        ( (eq elem (car lst))  (del elem (cdr lst)) )
        (t (cons (car lst) (del elem (cdr lst))))))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  'Less' handles the query with '<' as its predicate~~
~~  operator.  Given a query, property-name and       ~~
~~  property-value specified in the query, and a      ~~
~~  stream of entries, 'less' returns a filtered      ~~
~~  stream of entries which contain propertyvalues    ~~
~~  satisfying the query-condition.                   ~~
~~  'Mw-less' is called for the query with sm, rm,    ~~
~~  rw, and sw  as its property-name.                 ~~
~~  'Num-less' is for the query with rd, sd, and sy   ~~
~~  as its property-name.   'St-less' is for the query~~
~~  with ry.                                          ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun less(q pty value frame)
  (if (eq q 'error) 'error
  (cond ( (memq pty '(rm sm rw sw))
          (mw-less pty frame (get-list2 pty value)) )
          ( (memq pty '(rd sd sy))
            (num-less  pty value  frame))
        ( (eq pty 'ry )
          (st-less pty
                   (get_pname (concat value "")) frame))
        (t  'error) )))



(defun mw-less ( field frame lst)
  (cond ( (null lst) '())
        (t (mergel (equatel  field (car lst) frame)
                   (mw-less field frame (cdr lst)))) ))

                                                        '

(defun num-less( field qvalue l)
  (prog (frame prop-value)
   loop
   (setq prop-value (get-property-value field (car l)))
   (if  (< prop-value qvalue)
        (setq frame (cons (car l) frame)) )
   (setq l (cdr l))
   (if (null l) (return frame) (go loop) )  ) )
```

```
(defun st-less(field qvalue l)
  (prog (frame prop-value)
   loop
   (setq prop-value (get-property-value field (car l)))
   (if (alphalessp prop-value qvalue)
       (setq frame (cons (car l) frame)) )
   (setq l (cdr l))
   (if (null l) (return frame) (go loop) ) ))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  'Less-eq' handles a query with '<=' as its predicate~~
~~  operator.                                            ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun iess-eq (q entries)
  (mergel (less  q  (car q) (caor q) entries)
          (equate (nth 0 q) (nth 1 q) entries)))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~  'Greater' is a function to handle the query with      ~~
~~  '>' as its predicate operator.    For the query with ~~
~~  'rm, sm, rw, and sw' as its predicate-name, 'mw-gr'  ~~
~~  is called.  The query having 'rd, sd, and sy' is     ~~
~~  handled by a function, 'num-gr'.  'St-gr' takes care ~~
~~  of the query with 'ry'.                              ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun greater(q field value frame)
  (if (eq q 'error) 'error
  (cond ( (memq field '(rm sm rw sw))
          (mw-gr field frame (get-list4 field value)))
        ( (memq field '(rd sd sy))
          (num-gr field value frame) )
        ( (eq field 'ry )
          (st-gr field
            (get_pname (concat value "")) frame) )
        (t 'error) )))


(defun mw-gr (field frame lst)
  (cond ( (null lst) '())
        (t (mergel (equatel  field (car lst) frame)
                   (mw-gr field frame (cdr lst)))) ))


(defun num-gr( field qvalue l)
  (prog (frame prop-value)
```

```
    loop
    (setq prop-value (get-property-value field (car l)))
    (if  (>  prop-value qvalue)
         (setq frame (cons (car l) frame)) )
    (setq l (cdr l))
    (if (nu(l l) (return frame) (go loop) )  ))


(defun st-gr(field qvalue l)
  (prog (frame prop-value)
   loop
   (setq prop-value (get-property-value fie(d (car ())))
   (if (and (not (alphalessp prop-value qva(ue))
            (not (equa( prop-value qvalue)))
       (setq frame (cons (car l) frame)) )
   (setq l (cdr l))
   (if (nu(l l) (return frame) (go loop) ) ))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~   The query having '>=' as its predicate-operator     ~~
~~   is handled by 'greater-eq'.                         ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun greater-eq (q frame)
  (mergel (greater  q (car q) (cadr q)  frame)
          (equate (nth 0 q) (nth 1 q)  frame)))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~   'Get-list2' is a function called by a function,    ~~
~~   'less'.  Given a property-name and a month or      ~~
~~   week, 'get-list2' calls 'get-(istl' to obtain     ~~
~~   the (ist of months/week-names which are (ess than ~~
~~   given month or week and returns the (ist.         ~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

(defun get-listl (element lst)
  (if  (eq (car (st) e(ement)  '()
       (cons (car (st) (get-listl e(ement (cdr lst)))))))


(defun get-list2 (field value)
  (cond ( (memq field '(rw sw))
          (get-((stl va(ue (weeklist)))
        ( (memq field '(rm sm))
          (get-listl value (monthlist) ))))


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~   'Get-list4' is a function called by 'greater'.     ~~
```

```
;;  Given a property-name and a month or week name,     ;;
;;  this function calls 'get-list3' to obtain a list    ;;
;;  of months/weeknames which are greater than given    ;;
;;  month/week name, and returns the list.              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-list3 (element lst)
  (if  (eq (car lst) element) (cdr lst)
       (get-list3 element (cdr lst))))


(defun get-list4 (field value)
  (cond ( (memq field '(rw sw))
          (get-list3 value (weeklist)))
        ( (memq field '(rm sm))
          (get-list3 value (monthlist)))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  'Equatel' is a subfunction called by 'mw-gr' or     ;;
;;  'mw-less' to obtain a list of entries which         ;;
;;  contains the given property-values (qvalue) under   ;;
;;  the given properties (field) such as rw, rm, sw,    ;;
;;  and sm.                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun equatel( field qvalue l)
  (prog (frame prop-value)
   (cond ( (eq field 'sw)
           (setq qvalue (concat qvalue ",")))))
    loop
    (setq prop-value (get-property-value field (car l)))
    (cond ((eq  prop-value qvalue)
           (setq frame (cons (car l) frame)) ))
    (setq l (cdr l))
    (if (null l) (return frame) (go loop)  )))



(defun q-field-list()
  '(rdate sdate subject status to from body ) )


(defun monthlist()
  '(jan feb mar apr may jun jul aug sep oct nov dec) )


(defun weeklist()
  '(sun mon tue wed thu fri sat) )


(trace funcall)
```

```
~~~~  for auto loading      ~~~~
(setq user-top-level 'start)
```

```
#include <ctype.h>
#include <stdio.h>
#include <pwd.h>
#include <utmp.h>

char    *my_name;
char    *getlogin();
struct  passwd  *getpwuid();
int i, j;


ml(a)
   char a[];
{

        my_name = getlogin();
        if (my_name == NULL || strlen(my_name) == 0) {
                struct passwd *pwent;
                pwent = getpwuid(getuid());
                my_name = pwent->pw_name;               }

        j = 0;
        for (i=0; my_name[i]; i++)
          a[j++] = my_name[i];
        return(j);
   }
```

;

A QUERY SYSTEM FOR INFORMATION RETRIEVAL IN A MAILBOX

by

SONG HEE KIM

B.A., Ewha University, Seoul, Korea, 1976

AN ABSTRACT OF A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

ABSTRACT

An eletronic mailbox system can organize messages into a database according to their contents and provide users with a facility for categorizing desired message(s) for retrieval. The Query system for Information Retrieval in a Mailbox (QIRM) has been designed based upon such concepts.

QIRM increases the selectivity of information retreived by allowing users to specify their requests using a mail-query language resembling SQL. Also, QIRM provides some insights for developing an intelligent Unix mail facility by employing several techniques of Artificial Intelligence such as semantic networks, property lists, and matching. The database in QIRM is based on the semantic network structure and also utilizes an indexing scheme for the purpose of fast searching. The mail-query language is based on the tuple calculus and provides a user-friendly interface and a pattern-directed access of the messages. The query processor of QIRM employs the network-fragment-matching mechanism and utilizes the stream of property lists.