L-EQUEL: AN EMBEDDED QUERY LANGUAGE FOR FRANZ LISP

by

ANNE ROBERTA TRACHSEL

B.S., The Ohio State University, 1979

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by:

Roger T. Hartley
Major Professor

CONTENTS

ILLEGIBLE DOCUMENT

THE FOLLOWING DOCUMENT(S) IS OF POOR LEGIBILITY IN THE ORIGINAL

THIS IS THE BEST COPY AVAILABLE

1. CHAPTER ONE - INTRODUCTION

1.1 Overview

This master's report describes the design and implementation of the LISP-Embedded Query Language, L-EQUEL. EQUEL, Embedded Query Language, enables the C language programmer to embed INGRES queries within a C language program. L-EQUEL, a set of EQUEL-based database access routines for LISP, enables the Franz LISP programmer to embed INGRES database queries within a Franz LISP program.

Current LISP databases depend on features unique to LISP; the information stored in them is not accessible to programs written in other languages, because the databases are enclosed within the program's storage area. By allowing access to the INGRES database from within a LISP program, L-EQUEL provides a facility for data sharing between programs written in Franz LISP and programs written in other languages.

Franz LISP, developed at the University of California at Berkeley[1], is available on Berkeley UNIX* systems. INGRES and EQUEL, developed at University of California at Berkeley

---

* UNIX is a registered trademark of AT&T Bell Laboratories

[7] [9], are marketed by Relational Technology, Inc. [4]. INGRES is available for UNIX and VMS** operating systems [4] [6]. EQUEL was implemented using a combination of YACC (Yet Another Compiler Compiler) [12] and C language routines.

1.2 Contents of This Report

Chapter One of this report is this chapter. Chapter Two discusses current databases for LISP systems and contains an investigation of the feasibility of creating Franz LISP access routines for existing databases. Chapter Three presents the details of the design and implementation of the L-EQUEL database interface. Chapter Four discusses the results of the implementation. Chapter Five presents a summary of the implementation, and Chapter Six contains the acknowledgements. The references and appendices follow chapter six; the appendices contain a summary of the L-EQUEL Syntax, a manual page for L-EQUEL, an example of L-EQUEL use, and the Franz LISP code that implements L-EQUEL.

** Trademark of Digital Equipment Corporation.

2. CHAPTER TWO - BACKGROUND INFORMATION

This chapter describes current database features for LISP and provides examples of existing embedded query language systems.

2.1 Current Database Features For LISP

Current database features for LISP rely on capabilities unique to the LISP language, and the data is stored within the LISP program environment. Instead of maintaining the data separately on disk, the entire program image, including the data, is stored together.

Internal LISP databases tend to contain rather complex tuples, and to have relatively few occurrences of those tuples. In contrast, stand-alone database managements systems tend towards simpler individual tuples, and the databases contain many occurrences of those tuples.

2.1.1 Advantages

Relying on unique LISP features allows the database management system to maximize all possible internal efficiencies. The database doesn't have to support different languages' interfaces, so the database designer can tune the database for a particular application or set of applications.

2.1.2 Disadvantages

While the advantages listed above are valuable, the fact that existing LISP databases rely on internal LISP capabilities also limits the LISP programmer. He/she cannot access commercial, general-purpose database management systems. Also, the relative complexity of the tuples tends to make them more difficult to create and maintain.

2.1.3 PEARL - An Example

PEARL (Package for Efficient Access to Representations in LISP) is an example of a current internal LISP database system [11]. Developed at University of California in Berkeley, PEARL is an artificial intelligence language providing capabilities for manipulating associative databases. PEARL functions, which include database insertion and deletion, are compiled and loaded directly into LISP.

With PEARL, a programmer can insert and delete items from a database. The PEARL user manipulates "symbols" and "structures". "Symbols" are described as semantically equivalent to LISP atoms, but are used and represented differently for efficiency [7].

Although PEARL contains functions that could be used for general-purpose applications, it does depend on LISP's internal structure, so the restrictions listed above apply.

The author could find no example of an internal LISP database management system that provides data accessibility to programs written in other languages.

2.2 LISP Access to General-Purpose Databases

2.2.1 Desired General Database Features

In determining the feasibility and advantages of adding a Franz LISP interface to a given existing database management system, several criteria were considered. The database management system should:

- be general-purpose,
- be able to access data from within a host language program,
- provide exception handling,
- be flexible,
- allow the use of variable names for data values,
- provide a consistent access syntax, whether the queries are made from within a host-language program or are executed separately,
- be available for a UNIX system.

The next sections of this report discuss these criteria with respect to two existing database management systems and their associated embedded languages.

2.3 Existing Embedded Query Languages

The following sub-sections describe two existing relational database management systems, both of which feature embedded query languages. The database management system described first, System R, was developed by IBM. The second database management system discussed, INGRES, was developed at the University of California, Berkeley.

The discussion begins with general facts regarding each database management system. Following that, several features of the database management systems and their embedded languages are compared.

2.3.1 System R - Overview

System R began as an experimental relational database management system, developed at IBM Research Laboratory in San Jose, California [14][15]. After installation and evaluation at several IBM locations for more than two years, System R was adapted for commercial use by IBM Programming Center in Endicott, New York. The resulting product, called "SQL/Data System", was announced in 1981. SQL/Data System runs on the DOS/VSE operating system.

In this report, the terms "System R" and "SQL/Data System" are used interchangeably, except where noted.

The goals of System R development included the desire to support one-of-a-kind ("ad hoc" [14]) queries as well as those queries that can be defined once and executed repeatedly. While the latter form of query can be compiled into library routines, the former query type implies user interaction and thus must be handled separately during execution.

The decision was made that System R should support both a stand-alone query interface and an embedded language interface. The stand-alone interface was called UFI, the User-Friendly Interface. The embedded language features are available for COBOL and for PL/I. The user interface, called SQL, is consistent across the stand-alone and embedded language versions. As expected, user reaction to the consistent interface is favorable [15].

2.3.2 INGRES - Overview

INGRES is a relational database management system developed at the University of California in Berkeley, California, and marketed by Relational Technology, Inc. It runs on the UNIX operating system and is written primarily in C language.

INGRES provides a shell-level query interface known as QUEL (Query Language), and features EQUEL (Embedded Query Language). EQUEL enables the programmer to embed queries inside a C language program, providing the full capabilities of both C and QUEL. The EQUEL interface presented to the programmer is independent from EQUEL's interface to INGRES. Since the EQUEL command syntax is quite similar to the QUEL syntax, the programmer works with a consistent interface for database transactions whether they are writing a C language program or executing queries from the shell (QUEL).

2.3.3 Logistics Of The Embedded Interface

This section discusses the methods of user interaction with the embedded interface; that is, the appearance of embedded statements in the code and the invocation of the embedded language system. The section presents a discussion of System R's SQL logistics, and compares them with the logistics of the INGRES/EQUEL system.

For "canned interactions" (those queries that can be fully defined when the program is written), the SQL statements are embedded in the host COBOL or PL/I program. A preprocessor (called XPREP in System R [14]) recognizes the SQL statements because they are prefixed with a dollar sign ($), and compiles them into a series of machine-language routines. The machine-language routines collectively are called an "access module" [14][15]. In the user's program, host language calls to the access module replace the SQL statements, and the program can then be compiled normally. When the COBOL or PL/I program is executed, the access module provides the database interface.

With this approach, the error checking, selection of access path, and parsing are concentrated into the preprocessor steps and do not affect execution time. In addition, since the access module is tailored to the individual program's database needs, the running program interacts with a customized subset of the SQL capabilities that is smaller and executes more quickly than the complete SQL.

For interactive embedded queries, the parsing, validity-checking, and selection of the most appropriate access path must take place during execution, since the complete information is not available during preprocessing. However, experiments have illustrated that compiling the SQL statments into an access module is still efficient for these queries [14].

The treatment of EQUEL commands inside a C language program is similar in appearance to SQL, in that all EQUEL commands must be prefixed by two number signs (##). In addition, declaration statements for any C language variables used by EQUEL statements must be prefixed by two number signs. If the programmer wants a block of code to be executed repeatedly as a result of an EQUEL query, then the beginning and ending brackets ({}) of that block of code must be preceded by two number signs.

As in SQL, the EQUEL preprocessor recognizes the two number signs. Rather than replacing the preprocessor statements with machine-language access modules, EQUEL replaces the EQUEL statements with calls to the C language subroutines that provide the direct interface to INGRES. Any EQUEL commands' references to variables are included as parameters to these subroutine calls. The program can then be compiled with the C compiler, and loaded with the INGRES library to resolve the EQUEL/INGRES routine references.

The major difference in the logistics category between EQUEL and SQL occurs when the program originally containing the embedded statements is executed. The EQUEL/INGRES interface always interprets the database requests, while SQL executes compiled database requests. While the compilation is more efficient, interpretation allows more flexibility. For an interpreted interface, the binding time of the variables in the request can be delayed until execution; for a compiled interface, the binding occurs during compilation. Delaying the binding allows the C/EQUEL programmer to alter the domain and relation names at execution time, while the SQL programmer must rewrite source code to effect the same changes, because domain and relation names cannot be variables in SQL.

As mentioned in Section 2.3.1 of Chapter 2, SQL does provide the capability to execute interactive queries from the program; these queries expect a string argument that is interpreted at run time. However, this is still more complicated than EQUEL's method.

2.3.4 Variables

This section discusses the host-language variables used with EQUEL and SQL. Topics discussed include compilation-time versus execution-time binding of variables, type-checking and conversion, and restrictions on variables. As in the previous section, the discussion of each item begins with the SQL approach, and continues with a comparison of SQL to EQUEL.

2.3.4.1 Variable Restrictions

For SQL, variables represent data values, but may not replace table names or field names. In INGRES' EQUEL, variables may represent relation names, domain names, target list elements, or domain values.

2.3.4.2 Tuple Variables

A tuple variable is one which applies to all rows of a relation, and indicates a specific row at any particular time. In SQL, tuple variables are only present when needed to resolve ambiguous names, while in EQUEL tuple variables are always present. Tuple variables add some minor complexity to easy queries, but they enhance the readability of complicated queries.

2.3.4.3 Type-Checking and Conversion

For both SQL and EQUEL, variable type-checking and conversion are implicit; that is, the programmer does not specify anything to cause the checking and/or conversion to occur.

2.3.4.4 Copying Variables

For both EQUEL and SQL, values from a database record must be copied individually; that is, the programmer cannot copy an entire record with just one statement.

2.3.5 Side Effects

This section discusses the side effects associated with SQL and EQUEL. First the section contains a general discussion of side effects' advantages and disadvantages, then it compares the side effects of SQL and EQUEL.

The question of whether a system or language ought to have side effects is a source of controversy. Having side effects relieves the user from having to specify as many data manipulation commands; however, a system having side effects can be more difficult to debug. In addition, transactions are less easily understood when everything is not stated explicitly.

SQL has database procedures called "triggers"[13] which produce side effects. These "triggers" are present for queries involving READ, INSERT, DELETE, and UPDATE. The trigger is executed once for each tuple, and it may in turn force additional updates, based on the dependencies of the transaction.

EQUEL has no side effects.

2.3.6 Error Handling

When using SQL, the programmer must explicitly test a return code to determine whether an error has occurred. EQUEL, meanwhile, provides automatic notification when errors are encountered.

2.4 Choosing EQUEL For LISP

After surveying the literature and comparing existing embedded query languages, the author decided to base the LISP embedded interface on EQUEL. Several factors influenced this decision:

- EQUEL allows more extensive use of variables, and more explicit error-handling.

- EQUEL runs on Berkeley UNIX systems, and is available here at Kansas State University.

- There are defined mechanisms for communicating between C routines and Franz LISP routines [2] [3]. Therefore, there is an underlying compatibility between Franz LISP and EQUEL, since Franz LISP's kernel is written in C, and EQUEL is written in and for C.

- The syntax of EQUEL statements can be adapted to a LISP-compatible format with only minor alterations, thus maintaining the INGRES philosophy of providing a consistent database interface.

3. CHAPTER THREE - IMPLEMENTATION SECTION

This chapter discusses the implementation of the LISP Embedded Query Language (L-EQUEL). L-EQUEL provides features of the INGRES database's EQUEL for Franz LISP programs.

The chapter begins with a description of the goals and requirements for L-EQUEL. It continues by discussing the assumptions made, then describes the interfaces between L-EQUEL and INGRES and between L-EQUEL and the LISP programmer (L-EQUEL user). Finally, the high-level design and detailed-level design of L-EQUEL are discussed.

3.1 Implementation Goals and Requirements

The goals of the implementation are:

- The L-EQUEL commands will be embedded within Franz LISP programs.

- L-EQUEL will allow general-purpose queries.

- L-EQUEL will interact with the existing INGRES database just as EQUEL does.

- L-EQUEL syntax will be consistent with EQUEL syntax. The syntax differences between the two languages will be limited to those changes required for LISP compatibility.

3.1.1 Choice of Command Set

The command set for the initial version of L-EQUEL should meet the following criteria:

- Together, the commands allow the programmer to perform basic INGRES database capabilities. The basic capabilities consist of:
  - Opening a database,
  - Creating database tables,
  - Appending items to database tables,
  - Deleting items from tables,
  - Removing database tables,
  - Changing items' values in tables,
  - Retrieving information from tables,
  - Printing information to the terminal or to a file,
  - Closing a database.
- These commands are likely to be performed regularly.
- These commands are likely to require information found within the LISP program.

The following set of EQUEL commands is intended for the initial version of L-EQUEL. The commands are listed in alphabetical order. For a description of these commands' capabilities and EQUEL syntax, see references [4] and [5].

- Append
- Create
- Delete
- Destroy
- Exit
- Ingres
- Print
- Range
- Replace
- Retrieve

The following set of EQUEL commands should be added for a second version of L-EQUEL. While these commands are not necessary for the basic feature list, they provide additional convenience.

- Copy

- Integrity

- Set

3.2 Assumptions

In designing and implementing L-EQUEL, the author made the following assumptions:

- L-EQUEL will be implemented for and in Franz LISP.

- L-EQUEL assumes that the database to be manipulated has already been created. This is an assumption made by EQUEL as well; EQUEL does not allow the embedding of the INGRES QUEL "createdb" command to create a database.

- Similarly, L-EQUEL will not allow the programmer to embed the command for destroying a database (destroydb). The "destroy" command listed above pertains to destroying tables (relations) within a database.

- The version of INGRES / EQUEL with which L-EQUEL interfaces will be compatible with version 7.10, dated October 27, 1981. This is the version of INGRES in operation at Kansas State University at the time this report was written.

3.3 Interfaces

This section provides details of the interfaces of L-EQUEL.

3.3.1 User (Programmer) Interface - Commands

As mentioned in the goals and requirements above, L-EQUEL command syntax will basically be the same as the syntax for EQUEL. The only syntactic changes will be made for LISP syntax compatibility.

Appendix One lists the syntax of each of the L-EQUEL commands. Descriptions of the EQUEL commands on which they are based can be found in reference [4].

3.3.2 User Interface - Invoking L-EQUEL

Figure One illustrates the progression from the input file, containing both LISP and L-EQUEL statements, through the preprocessor to the output file, which contains just LISP source code.

```
LISP and L-EQUEL ----> L-EQUEL Preprocessor ----> LISP
(nfile.lq)                                        (nfile.l)
```

__Figure 1__

L-EQUEL Input File to Output File

To invoke L-EQUEL, the LISP programmer executes the command "lequel". The command "lequel" takes one argument: the name of the Franz LISP source file containing the embedded L-

EQUEL statements. The command produces a file containing the Franz LISP source, with the L-EQUEL statements replaced by calls to the C language subroutines that comprise the L-EQUEL/INGRES interface. The resulting Franz LISP file can then be compiled and loaded into LISP. Appendix II contains the manual page for "lequel".

The file-naming conventions required by lequel are listed below:

- the name of the input file containing both the Franz LISP and the L-EQUEL statements must end in ".lq".
- The name of the output file produced by L-EQUEL will be the same as the name of the input file, except that a ".l" will replace the ".lq" of the input file name.

Example A:

The file named "fred.lq" is a Franz LISP source file containing embedded L-EQUEL statments. The following command produces an output file named "fred.l":

```
lequel fred.lq
```

3.3.3 Software (INGRES) Interface

The L-EQUEL system interface to INGRES is modeled after the EQUEL / INGRES interface rather than the C / EQUEL interface. The C / EQUEL interface is advertised as supported, so that future versions of the C / EQUEL interface are likely to be compatible with the current interface. The EQUEL / INGRES interface is not advertised as supported.

Consideration was given to meeting the C / EQUEL interface with L-EQUEL. However, passing pieces of L-EQUEL through EQUEL would have required rewriting LISP statements in C. This would be necessary since related L-EQUEL statements may enclose a section of host-language code. For example, if several tuples are to be retrieved, often there is a set of statements to be executed for each tuple. These statements appear between the beginning L-EQUEL retrieve command and the part of the L-EQUEL retrieve command that ends the retrieval.

The decision was made not to pursue meeting the C / EQUEL interface.

The EQUEL / INGRES interface consists of a series of C language routines. Franz LISP provides the capability to load compiled C language routines directly into a LISP program environment, using the LISP "cfasl" and "getaddress" routines. The L-EQUEL preprocessor will replace the L-EQUEL statements with calls to EQUEL / INGRES interface routines. The calls to C language routines in LISP maintain the LISP syntax style; that is, the function name is listed first, followed by the function's parameters.

3.4 The Design Of L-EQUEL

This section describes the design of L-EQUEL. The section begins by specifying the high-level architecture of L-EQUEL. It continues by discussing individual modules in greater detail, and specifies what data structures are required.

3.4.1 Modules of L-EQUEL

The design of L-EQUEL is composed of several software modules, as illustrated by Figure 2.

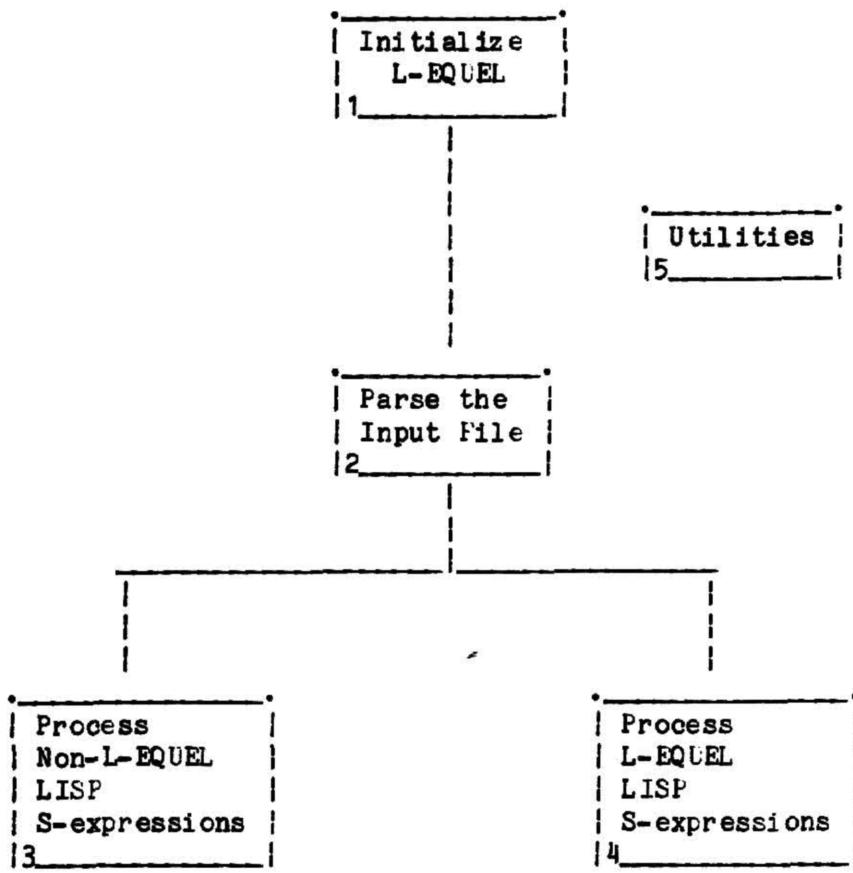


Figure 2

The L-EQUEL Modules

First, L-EQUEL must perform initialization functions, such as verifying the existence of the source file and setting up tables. Next, L-EQUEL must parse the source file, looking for occurrences of the special "%%" symbol. Third, L-EQUEL must process the non-L-EQUEL s-expressions, and finally, L-EQUEL must process the L-EQUEL s-expressions. A fifth module, the utility module, contains routines used throughout the L-EQUEL program. The following paragraphs discuss each of these modules in greater detail.

3.4.1.1 Initialization

Figure 3 illustrates the initialization section.

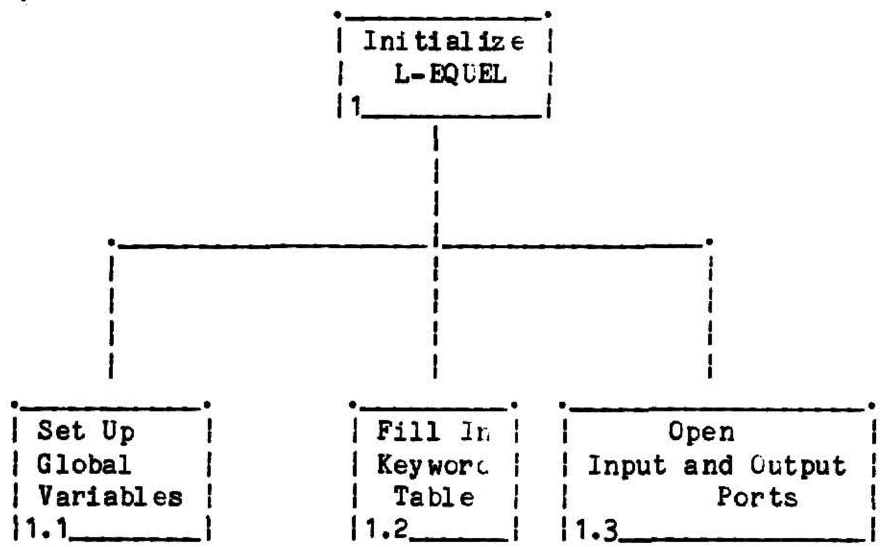


Figure 3

L-EQUEL Initialization Module

First, the global variables are initialized. One set of globals pertains to error messages. To enable these messages to be consistent throughout the code, global strings are defined for specific error messages. See the section on error handling for the list of error types for which messages are specified.

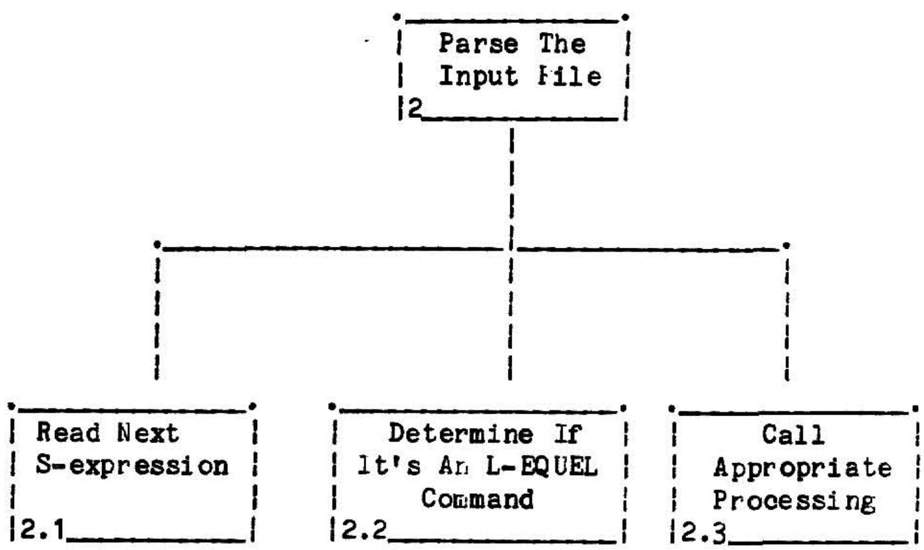
A second category of global variable is intended for debugging L-EQUEL during development. By setting the variable "lequeldebug" to a specific value, the printing of a series of debugging statements is controlled. For details regarding this, see the "Utilities" design section.

Next, the keyword table is initialized. This table's contents are discussed in section 3.4.3 of this chapter.

Finally, the ports for the input and output files are opened.

3.4.1.2 Parsing The Input File

As shown in Figure 4, the parsing module reads the input file, one s-expression at a time, and determines whether the expression contains any L-EQUEL commands.



<u>Figure 4</u>

L-EQUEL Parsing Module

The parser is recursive; that is, s-expressions may be nested. When a new s-expression is read, its first member is examined:

- If the first member is "%%", the parser sends the s-expression to "Processing L-EQUEL Commands".

- If the first member is some other atom, the entire list is sent to a separate "atom-parsing" routine, called "parseatom". (This routine is part of the "parsing" module.)

- If the first member of the s-expression is an s-expression, then the "parse s-expression" (parseexp) routine calls itself.

The "atom-parsing" routine examines the first member of the list it receives. If the first member is an atom, "parse-atom" writes it to the output file. (If the first member is not an atom, it is an error.) Then "parse_atom" examines the next member of the list it received. If this member is an atom, then "parse-atom" calls itself with the "cdr" of the original list. If the second member of the original list is an s-expression, then "parse-atom" calls "parse s-expression" with the "cdr" of the original list.

The "s-expression parsing" routine examines the first member of the list. If the first member is an atom, then the entire list is sent to the "atom-parsing" routine. If the first member is an s-expression, "parse s-expression" calls itself with the first member of the original s-expression..

Then the second member of the original list is examined. If this member is an atom, then "parse-atom" is called with the "cdr" of the original list. If the second member of the original expression is a list, then "parse-expression" calls a second expression-parsing routine, "parse2exp".

Parse2exp calls "parse-atom" or "parse-expression", depending on the first member of the list it receives. Parse2exp should only be called if the "car" of the original s-expression was a list.

3.4.1.3 Processing Non-L-EQUEL Commands

This module, illustrated in Figure 5, simply copies all non-L-EQUEL s-expressions directly to the output file.

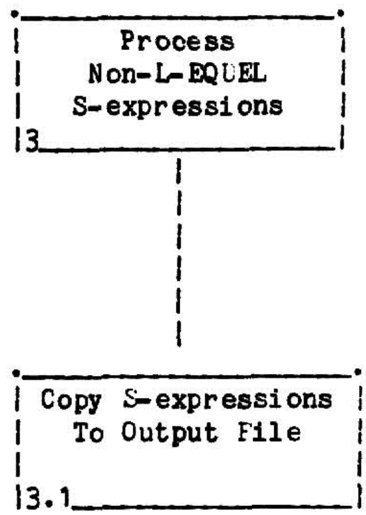


__Figure 5__

L-EQUEL "Process Non-L-EQUEL S-Expressions" Module

This step is best considered as a separate module for future expandability; however, for simplicity it can be implemented as a direct extension of the parsing step.

3.4.1.4 Processing L-EQUEL Commands

This module contains more levels of information than the previous modules. Its top level is shown in Figure 6.

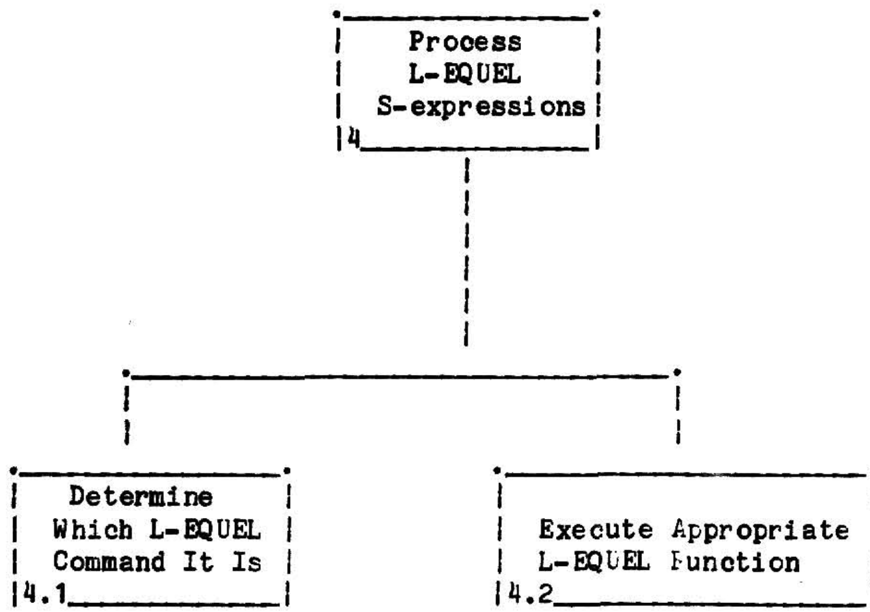


Figure 6

L-EQUEL "Process L-EQUEL S-Expressions" Module

First, the module locates the L-EQUEL command in the keyword table to determine what function to call. Then, the corresponding command function is called.

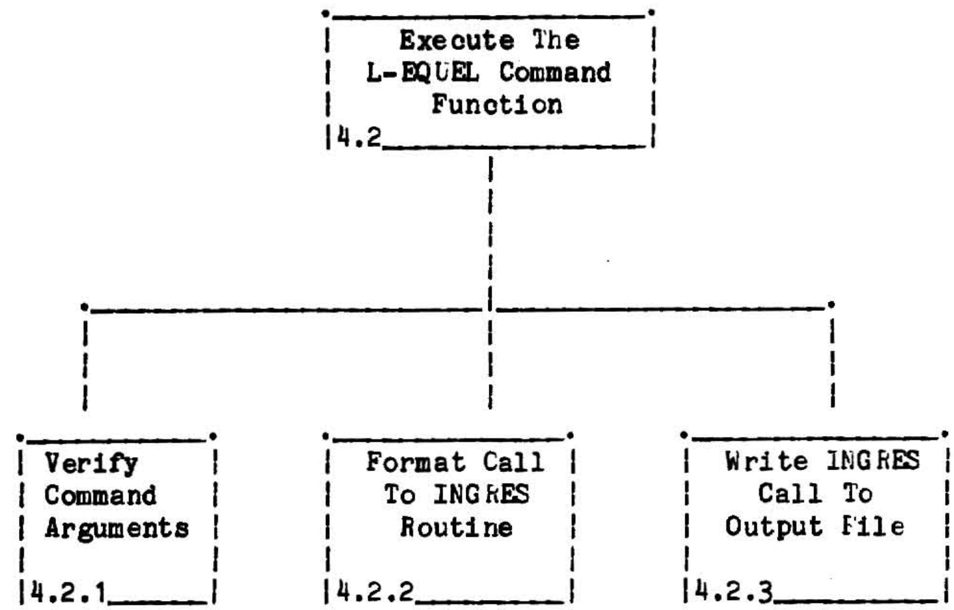


Figure 7

L-EQUEL "Execute The L-EQUEL Command Function" Sub-Module

As seen in figure 7, the command's arguments are verified, and the call to the appropriate C-language INGRES interface routine is formated into an s-expression. Finally, the s-expression containing the call to the INGRES interface routine is written to the output file.

3.4.1.5 Utilities

The design of L-EQUEL includes several utilities. Figure 8 illustrates them, and the following sections discuss their functions.

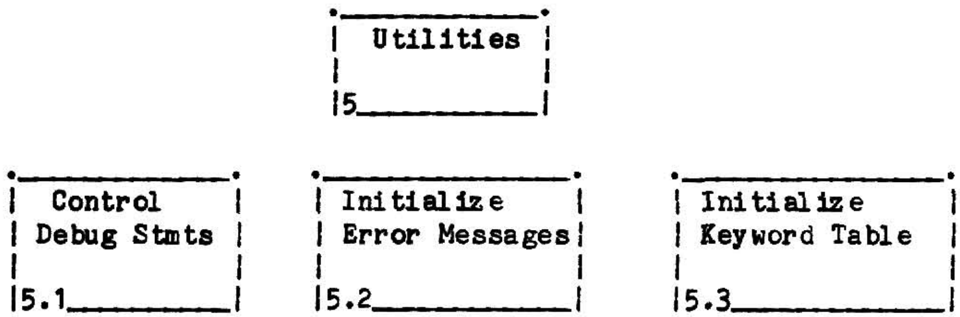


Figure 8

The L-EQUEL Utilities Module

Figure 8 does not have connecting lines as in the earlier figures because the utilities together have no notion of time-ordering. The utilities are grouped into a module because they perform specific, lower-level tasks, and may relate to more than one of the other modules.

3.4.1.5.1 Debugging Statements

The first set of utilities controls the printing of debugging statements on the user's terminal. If the value of the variable "lequeldebug" is zero, these utilities are defined to do nothing. When "lequeldebug" has a value of one, these utilities call normal LISP printing routines. This allows the debugging statements to remain in the source code without greatly affecting the running L-EQUEL program.

3.4.1.5.2 Error Messages

Another utility routine sets up the global variables for the error message strings. As mentioned earlier, these error message strings are global to promote consistency of error messages throughout L-EQUEL.

3.4.1.5.3 <u>Keyword Table</u>

One utility is responsible for initializing the table of valid keywords. The format and use of this table are explained in section 3.4.3 of this chapter.

3.4.2 The Lequel Command - Internal Viewpoint

The command "lequel" is created using the LISP compiler's "-r" option. This "autorun" option allows the resulting LISP program to be run separately, instead of requiring the Franz LISP compiler or interpreter to be invoked.

3.4.3 The Keyword Table

To recognize the L-EQUEL commands, a keyword table is maintained in the form of a LISP list. Each keyword name is an item on the list, and each item has a property called "op_function" associated with it. The op_function property's value contains the name of the function that is to be called when the associated keyword name is encountered in the source file.

3.4.4 Treatment of Variables

Where possible, the "type" (character string, integer, floating-point number) of the input variable will be verified for correctness. Type conversion will not be done for the initial version of L-EQUEL.

3.4.5 Use Of INGRES Library

To call the INGRES library routines from inside the "lequel" program, those routines' object files must be loaded into "lequel" using the "cfasl" and "getaddress" functions. The source program that contains embedded L-EQUEL statements must load in the appropriate INGRES interface routines.

3.5 Detailed Design Of L-EQUEL Command Functions

This section presents the detailed design of the functions for processing the specific L-EQUEL commands. The information presented for each function consists of the function's parameters, the return value, the tasks performed by the function, the algorithm used, and what unexpected events are handled by this function.

The commented code appears as Appendix IV.

3.5.1 Common Design Features

This section discusses the detailed design items that are common across several L-EQUEL command routines.

3.5.1.1 INGRES Interface

Each of the L-EQUEL command functions places in the output file a call to one of the EQUEL / INGRES interface routines. The names of all the interface routines begin with "II", and the description for each command specifies what interface routine call is appropriate for that command.

3.5.1.2 Command Options

Each L-EQUEL command has a (possibly null) set of valid options. To analyze those options, each L-EQUEL command function maintains a LISP list of the valid options. Currently these lists just contain the options' formats, but they could be expanded to include properties for each of the options.

3.5.1.3 Error Handling

If a function determines that an error has occurred, it writes a message to the terminal and writes the erroneous s-expression to the output file with a short message explaining the error. The error message formats are globally defined strings, so that the error messages will be consistent throughout the code.

The following types of error conditions are handled:

- Illegal Keywords

- Missing Table Name

- Invalid Parameters

- General Syntax Errors

3.5.2 The LEAPPEND Command

The "f_leappend" function writes a call to "IIwrite" to the output file, followed by a call to "IIsync". The syntax-checking consists of:

- Verifying the presence of the table name,

- Verifying that there is a target-list of columns and values to be appended,

- Verifying that there is a value field corresponding to each name field.

All other checking will be performed by INGRES.

3.5.3 The LECREATE Command

The function "f_lecreate" writes a call to "IIwrite" to the output file, followed by a call to "IIsync". The function makes the following syntax checks:

- The presence of a tablename field is checked,

- The presence of the list of field names and formats is verified,

- There must be an equal number of field names and formats in the list,

- The proposed field formats in the command's parameters are checked against the INGRES limits for field types and widths.

The field names are assumed to be valid.

3.5.4 The LEDELETE Command

The function "f_ledelete" writes to the output file a call to "IIwrite", followed by a call to "IIsync." The parameters to the command are assumed to be correct.

3.5.5 The LEDESTROYCommand"

The function "f_ledestroy" writes a call to "IIwrite" to the output file, followed by a call to "IIsync". The function verifies that at least one table name field is present.

3.5.6 The LEEXIT Command

The "f_leexit" function writes a call to "IIexit" to the output file. Since no arguments to dbexit are expected, no parameter-checking is performed by "f_leexit".

3.5.7 The LEINGRES Command

The "f_leingres" function writes a call to "IIingres" to the output file. The "leingres" from the command line is replaced with "IIingres". The validity of each option is verified by searching the list created during initialization. This routine does not check the options' argument values; it assumes that they are valid.

3.5.8 The LEPRINT Command

The function "f_leprint" writes a call to "IIwrite" to the output file, followed by a call to "IIsync". The function assumes that all arguments are valid INGRES table names, so its only syntax checking is to verify that at least one table name is present.

3.5.9 The LERANGE Command

The function "f_lerange" writes a call to "IIwrite" to the output file, followed by a call to "IIsync". The function verifies that the words "of" and "is" appear in the proper order, and that the range value and table name fields are also present.

3.5.10 The LEREPLACE Command

The "f_lereplace" function writes a call to "IIwrite" to the output file, followed by a call to "IIsync". The syntax-checking consists of:

- Verifying the presence of a range variable,
- Checking that the range variable is an atom,
- Verifying the presence of a target-list of column names and values to replace,
- Checking that the quantity of names matches the quantity of values in the target list.

3.5.11 The LERETRIEVE Command

The "leretrieve" command is the most complex of the initial set of L-EQUEL functions, because it is possible to have embedded LISP statements that are to be executed for each tuple retrieved. As a result, the retrieval depends on three separate L-EQUEL statements: "%% leretrieve", "%% leretrbgn", and "%% leretrdone".

The "f_leretrieve" is called when the "%% leretrieve" is encountered. It takes care of the initial processing of the retrieval. It places calls to "IIwrite", "IIsetup", "IIn_get", "IIn_ret", and "IIerrtest" in the output file. The "f_leretrieve" function also begins creation of the "prog" loop that is necessary to implement the looping feature of the retrieve. For syntax-checking, "f_leretrieve" verifies that all variables used in the "leretrieve" statement have been declared.

The "f_leretrbgn" is called when the "%% leretrbgn" is encountered. The "%% leretrbgn" must precede the embedded LISP code for the retrieval.

When the "%% leretrdone" is encountered, "f_leretrdone" is executed. This routine adds the final piece of looping code to the retrieval. The embedded LISP statements are placed between the "%% leretrbgn" and the "%% leretrdone". Since the embedded statements are not L-EQUEL statements, they are copied directly to the output file.

4. CHAPTER FOUR - IMPLEMENTATION RESULTS

This chapter discusses the results of the initial implementation of L-EQUEL. After specifying whether the goals and requirements were achieved, the chapter discusses the design issues and difficulties encountered. A description of possible extensions and changes completes the chapter.

4.1 Evaluation of Requirements

This section specifies whether the goals discussed in chapter three of this report were realized. Each goal stated in section 3.1 of that chapter is discussed separately.

The first goal is "The L-EQUEL commands will be embedded within Franz LISP programs". This goal has been satisfied, in that a set of basic INGRES capabilities is available from within Franz LISP.

The second goal is "L-EQUEL will allow general-purpose queries". This goal also was satisfied, since the INGRES database is a general-purpose, relational database.

The third goal is "L-EQUEL will interact with the existing INGRES database just as EQUEL does". Since the decision was made to meet the EQUEL/INGRES interface rather than the C/EQUEL interface, this goal has also been met.

The fourth goal is that "L-EQUEL syntax will be consistent with EQUEL syntax". This goal has also been achieved. There are two types of syntax differences, but they do not interfere with this goal. One set of differences occurs to enable L-EQUEL syntax to be consistent with LISP syntax. The second set of differences results from EQUEL options that are not implemented in the initial version of L-EQUEL.

4.2 Design Issues and Difficulties

This section evaluates the design issues and difficulties encountered in developing L-EQUEL.

4.2.1 Internal EQUEL Documentation

There was sufficient user-level documentation for INGRES and for EQUEL. However, the author was not able to find internal documentation for EQUEL; that is, documentation of EQUEL's interface to INGRES. As a result, the author determined this interface by reading the existing EQUEL code (which consisted of YACC, and fairly well-commented C).

A complication that arose as a result of this issue was that it was difficult to determine which INGRES interface routines should be called for each unique situation.

The author ran an example of each command through EQUEL, and looked at the resulting ".c" files to determine the proper INGRES interface routines. For the more complex commands, the author ran EQUEL on several examples, in an attempt to include examples of the different options.

4.2.2 Retrieve Command

The EQUEL "retrieve" command ("leretrieve" in L-EQUEL) proved to be the most complicated of the command set for the initial implementation of L-EQUEL. It is the only command that contains a block of related host-language code that is to be executed for each tuple. In addition, it requires more handling of host-language variables.

To implement the "looping" of the host language code, it was necessary to insert a LISP "prog" construct in the L-EQUEL output file.

4.3 Extensions and Suggested Changes

Each of the following subsections discusses a separate possible extension to this initial implementation of L-EQUEL.

4.3.1 Additional Utility Routine

Since the INGRES C-language library routines need to be loaded in whenever LISP code resulting from L-EQUEL is to be executed, it would be convenient to provide a LISP utility routine to load them. This would simplify the procedure for using L-EQUEL, because the L-EQUEL user would no longer have to be concerned with remembering the correct "cfasl" and "getaddress" calls.

4.3.2 Method Of Invoking The Command

Currently, "lequel" is executed as a separate command. There is an alternative method of invoking "lequel". One could load "lequel" into the current LISP compiler environment. In that situation, the "%%" becomes the "main" macro that causes the L-EQUEL routines to be executed.

One disadvantage to this alternative method of invocation is that debugging L-EQUEL would become more difficult. Since L-EQUEL is embedded into the LISP compiler, one could not always determine whether a bug results from L-EQUEL problems or from the compiler.

4.3.3 Additional EQUEL Command Capabilities

An obvious set of extensions would be to add additional EQUEL command command capabilities, as discussed in section 3.1.1 of Chapter Three of this report.

5. CHAPTER FIVE - SUMMARY

This report has described the implementation of L-EQUEL, a set of Franz LISP access routines to INGRES, a general-purpose, relational database. The implementation has adapted EQUEL, INGRES' embedded query language for programs written in C, for use inside Franz LISP programs. The implementation takes advantage of existing LISP functions designed to allow access to C language routines from within a Franz LISP program.

The L-EQUEL access routines provide an alternative to Franz LISP program developers who need to maintain data. Whereas these program developers previously were limited to maintaining an internal database for their applications, they now have the opportunity to access an existing, general-purpose, relational database. A program developer could choose to use both the internal LISP features and the L-EQUEL access routines for different phases of the same application.

6. CHAPTER SIX - ACKNOWLEDGEMENTS

I want to extend thanks to Dr. Roger Hartley, my major professor, for his support, and to the other members of my graduate committee, Dr. Elizabeth Unger and Dr. Virgil Wallentine, for their contributions. In addition, I want to thank all the participants, faculty, and staff of the AT&T Summer-on-Campus program at Kansas State University for helping to make my experiences here worthwhile.

BIBLIOGRAPHY

[1] Wilensky, Robert. (1984). LISPcraft. W. W. Norton & Company, Inc., New York, N.Y., c. 1984.

[2] Andreson, Fred P. (1984). "The Franz Lisp - C Interface". Computer Vision Laboratory Center for Automation Research, University of Maryland, College Park, Maryland. June, 1984.

[3] Foderaro, John K. and Sklower, Keith L. (April 1982). "The FRANZ Lisp Manual, A document in four movements". c. Regents of the University of California, 1980, 1981.

[4] "INGRES Reference Manual (Version 2.0, VAX/UNIX, June 1983)". Relational Technology Inc., Berkeley, California, c. 1983.

[5] "EQUEL/C User's Guide (Version 2.0, VAX/UNIX, June, 1983)". Relational Technology, Inc., Berkeley, California, c. 1983.

[6] "An Introduction To INGRES (Version 2.1, VAX/VMS, September, 1983) - Preliminary Version)". Relational Technology Inc., Berkeley, California, c. 1983.

[7] Stonebraker, Michael, Wong, Eugene, Kreps, Peter, and Held, Gerald. (1976). "The Design and Implementation of INGRES". Department of Electrical Engineering and Computer Science, University of California, Berkeley, California and Tandem Computers, Inc., Cupertino, California.

[8] Rowe, Lawrence A., "The INGRES Relational Database Management System". Relational Techonology, Inc., Berkeley, California.

[9] Stonebraker, Michael. (1980). "Retrospection on a Database System". ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980, pages 225-240.

[10] Allman, Eric and Stonebraker, Michael. (1982). "Observations on the Evolution of a Software System". Computer, June 1982.

[11] Deering, Michael, Faletti, Joseph, and Wilensky, Robert. (1982). "Using the PEARL AI Package (Package for Efficient Access to Representations in Lisp)". Computer Science Division, Department of EECS, University of California, Berkeley, Berkeley, California, February 1982.

[12] "UNIX(tm) System V, Release 2.0 - Support Tools Guide 307-108, Issue 2, April 1984". c. AT&T Technologies,

1984.

[13] Chamberlin, D.D., Astrahan, M.M, Eswaran, K. P., Griffiths, P.P., Lorie, R.A., Mehl, J.W., Reisner, P., Wade, B.W. (1976). "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control". IBM Journal of Research and Development, Volume 20, No. 6, November 1976.

[14] Chamberlin, D.D., Astrahan, M.M., King, W.F., Lorie, R.A., Mehl, J.W., Price, T.G., Schkolnick, M., Griffiths Selinger, P., Slutz, D.R., Wade, B.W., Yost, R.A. (1981). "Support for Repetitive Transactions and Ad Hoc Queries in System R". ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981.

[15] Chamberlin, Donald D., Gilbert, Arthur M., Yost, Robert A. (1981). IBM Research Laboratory, San Jose, California and IBM Programming Center, Endicott, New York. "A History Of System R and SQL/DATA System". Seventh International Conference on Very Large Data Bases, Cannes, France, September, 1981. [AO]

[16] Stonebraker, Michael and Rowe, Lawrence A. (1977). Department of Electrical Engineering and Computer Sciences Electronic Research Laboratory, University of California, Berkeley, California. "Observations On Data Manipulation Languages and Their Embedding In

General Purpose Programming Languages". Third International Conference on Very Large Data Bases, Tokyo, Japan, October, 1977. [AN]

APPENDIX I - L-EQUEL Command Syntax

This appendix summarizes the general syntax rules of L-EQUEL as they differ from the syntax rules of EQUEL. Following that, each L-EQUEL command is stated with its parameters' valid ranges and value types.

I.1 General Syntax Statements

The syntax of the L-EQUEL (LISP language Embedded QUEry Language) commands is similar to the syntax of the EQUEL commands. All of the L-EQUEL commands adhere to the following syntax guidelines:

1. They begin with "%%" instead of with "##".

2. The command is enclosed in a LISP list (s-expression).

3. The L-EQUEL command is always the first symbol following the "%%".

4. The name of an L-EQUEL command consists of the name of the corresponding EQUEL command, prefixed by "le".

5. Where EQUEL would state "param=pvalue", L-EQUEL will state "param pvalue" (separated by spaces instead of by an equals sign). This is consistent with LISP syntax, and eliminates the complication of parsing the "param=pvalue" construct. (The "param=pvalue" construct could contain one LISP atom or more than one

LISP atom, and the L-EQUEL parser would have to be sophisticated enough to handle both cases.)

As in EQUEL, all the keywords are reserved.

As in EQUEL, all the QUEL keywords are reserved.

I.2 L-EQUEL Command List

The following list contains the EQUEL and L-EQUEL versions of the commands present in the initial version of L-EQUEL. The L-EQUEL command syntax appears first on each line, followed by the EQUEL syntax.

- Leappend -- Append
- Lecreate -- Create
- Ledelete -- Delete
- Ledestroy -- Destroy
- Leexit -- Exit
- Leingres -- Ingres
- Leprint -- Print
- Lerange -- Range
- Lereplace -- Replace

- Leretrieve -- Retrieve

I.3 Notation

This section contains the key to the notation used in the syntax specifications below.

- The curly brackets ({}) enclose a construct that may appear zero or more times.

- The square brackets ([]) enclose a construct that may appear zero or one time.

- When nothing surrounds a construct, it is mandatory.

I.4 Individual L-EQUEL Command Syntax

For each L-EQUEL command, the syntax and explanation of parameters appears below. The EQUEL syntax for the command is included for reference.

The majority of the information regarding the command parameters is taken from reference [4].

I.4.1 LEAPPEND

I.4.1.1 Name

LEAPPEND - Append new rows to an existing table.

I.4.1.2 Calling Syntax

L-EQUEL:

```
(%% leappend [to] tablename (colname colvalue
             {colname colvalue}) [where qual] )
```

EQUEL:

```
## append [to] tablename (colname=colvalue
          {, colname=colvalue}) [where qual]
```

I.4.1.3 Description and Parameters

The dbappend command adds rows satisfying the "qual" qualifications to the currently active table.

The colname specifies which column names will have new values appended. The columns may be listed in any order.

The colvalue specifies what new value to append for that column.

One difference from the EQUEL version is that this implementation of L-EQUEL requires single values for the column values, rather than allowing expressions that evaluate to a single value.

For more information concerning LEAPPEND permissions, see reference [4]'s description of the APPEND command.

I.4.1.4 <u>Examples</u>

To append a new class to the "class" database:

```
(%% leappend "class" (cnumber 755 csubject "Computer_Science"
                    clocation "Nichols_125") )
```

I.4.2 LECREATE

I.4.2.1 Name

LECREATE - create a database.

I.4.2.2 Calling Syntax

L-EQUEL:

```
(%% lecreate tablename (columname format
                        {columname format} ) [with logging])
```

EQUEL:

```
## create tablename (columname = format)
        {,columname = format} [with logging]
```

I.4.2.3 Description and Parameters

This command enters a new table into the currently active database.

The tablename is the name of the table, and the columname(s) are the names of the columns.

The formats may be one of the following: "c1" to "c255" (character formats), "i1", "i2", "i4" (integer formats), "f4" , "f8" (floating-point formats), "date", or "money". Formats are described in Chapter 1, section 1.2.6 of reference [4].

Table limits, column limits, and table expiration information can also be found in reference [4].

I.4.2.4 Examples

To create a database calleo "class", with columns "csubject", "cnumber" and "clocation":

```
(%% lecreate "class" (csubject c20 cnumber i4
                 clocation c20) )
```

I.4.3 LEDELETE

I.4.3.1 Name

LEDELETE - delete rows from an existing table.

I.4.3.2 Calling Syntax

L-EQUEL:

```
(%% ledelete range_var [where qual] )
```

EQUEL:

```
## delete range_var [where qual]
```

I.4.3.3 Description and Parameters

The ledelete command removes rows that meet the qual qualifications from the table associated with range_var in the currently active database.

The range_var must either already exist, or it must be the default value for that table.

Note:

This is one case in which the LISP version still requires the EQUEL format for the qualification. This occurs because the qualification may include additional arithmetic operations besides the equals sign.

For additional caveats and permission information, see reference [4].

I.4.3.4 Examples

To remove a particular class from the database:

```
($$ lerange of "c" is "class")
($$ ledelete "c" where c.subject=
```

I.4.4 LEDESTROY

I.4.4.1 Name

LEDESTROY - Destroy existing tables.

I.4.4.2 Calling Syntax

L-EQUEL:

```
(%% ledestroy tablename)
```

EQUEL:

```
## destroy tablename
```

I.4.4.3 Description and Parameters

The ledestroy command removes tables from the database.

This command differs from ledelete in that ledelete deletes tuples from the tables, and ledestroy deletes the entire table.

I.4.4.4 Examples

To destroy a table named "class":

```
(%% ledestroy "class")
```

I.4.5 LEEXIT

I.4.5.1 Name

LEEXIT - terminate access to the current INGRES Database.

I.4.5.2 Calling Syntax

L-EQUEL:

(%% leexit)

EQUEL:

exit

I.4.5.3 Description and Parameters

This command terminates access to the currently active INGRES database. Once this command is given, another database may be made active by using the leingres command.

This command has no parameters.

I.4.5.4 Examples

To finish accessing the current database:

(%% leexit)

I.4.6 LEINGRES

I.4.6.1 Name

LEINGRES - allow access to an INGRES database by invoking the INGRES database system.

I.4.6.2 Calling Syntax

L-EQUEL:

($$ leingres [flags] dbname)

EQUEL:

ingres [flags] dbname

I.4.6.3 Description and Parameters

The ingres command invokes the INGRES database system. The dbname must be the name of a currently existing database.

See reference [4] for the flags options.

I.4.6.4 Examples

To invoke INGRES on the "academia" database:

($$ leingres "academia")

To invoke INGRES using flags for the "academia" database:

($$ leingres "-c8" "-i48" "academia")

I.4.7 LEPRINT

I.4.7.1 Name

LEPRINT - print information for one or more tables.

I.4.7.2 Calling Syntax

L-EQUEL:

(%% leprint tablename {tablename})

EQUEL:

print tablename, {tablename}

I.4.7.3 Description and Parameters

This command prints table information for the tables specified by tablename on the user's terminal, or on the standard output device. The format used for the printing may be set by the flags to the ingres command.

For restrictions on printing, see reference [4].

I.4.7.4 Examples

To print the information from the "class" table:

(%% leprint "class")

I.4.8 LERANGE

I.4.8.1 Name

LERANGE - Declare a variable to refer to specific rows in a specific copy of a table.

I.4.8.2 Calling Syntax

L-EQUEL:

```
($$ lerange of range_var is tablename )
```

EQUEL:

```
## range of range_var is tablename
```

I.4.8.3 Description and Parameters

This command declares a range variable that can be used in later INGRES statements. The range var is associated with tablename and refers to a specific copy of that table. The range variable remains in effect for the entire INGRES session, unless it is redeclared or the table is removed.

Note: in L-EQUEL, as in EQUEL, only one range variable can be specified per lerange statement. From QUEL, however, more than one range variable can be specified per call.

I.4.8.4 Examples

To define a range variable for the "class" table:

```
($$ lerange of "c" is "class")
```

I.4.9 LEREPLACE

I.4.9.1 Name

LEREPLACE - replace column values in a table.

I.4.9.2 Calling Syntax

L-EQUEL:

```
(%% lereplace range_var (target_list) [where qual] )
```

EQUEL:

```
## replace range_var (target_list) [where qual]
```

I.4.9.3 Description and Parameters

The lereplace command replaces values in the table specified by range var for rows that meet the qual qualification. The target list contains those columns and values that are to be replaced.

Note:

This is another case in which the EQUEL syntax is maintained, rather than the LISP syntax, for the qual portion of the command.

I.4.9.4 Examples

To replace the class number of Computer Science class "755" with a class number of "860":

```
($$ lerange of "c" is "class")
($$ lereplace c (cnumber 860) where
               c.csubject=
               c.cnumber=755 )
```

I.4.10 LERETRIEVE

I.4.10.1 Name

LERETRIEVE - retrieve rows from a table.

I.4.10.2 Calling Syntax

L-EQUEL:

```
(%% leretrieve (target_list) [where qual])
```

EQUEL:

```
## retrieve (target_list) [where qual]
```

I.4.10.3 Description and Parameters

This command retrieves all rows that satisfy the *qual* qualification and places them in a file or displays them on the standard output device.

Note:

this is another case in which the EQUEL format including the equals sign is used for *qual*.

For additional information, see reference [4].

I.4.10.4 <u>Examples</u>

To retrieve all classes that belong to the Computer Science department:

```
($$ lerange of "c" is "class")
($$ leretrieve (c.cnumber) where c.subject=
```

APPENDIX II - Manual Page for L-EQUEL Command

Command Syntax:

lequel <infile>

Description

This command invokes the L-EQUEL preprocessor on the file infile. The name of infile must end in ".lq".

The lequel command produces as its output a file having the same name as infile without the "q" on the end of the name. (The file name ends in ".l".)

The infile is assumed to consist of LISP s-expressions and L-EQUEL commands. The output file consists entirely of LISP s-expressions; the L-EQUEL commands are translated into LISP-format calls to the appropriate INGRES interface routines.

Caveats:

The preprocessor changes the format of the input file, so that spacing and carriage returns will probably not be as they were in the initial source file. The semantics of the file remain intact.

APPENDIX III - Example of L-EQUEL Use

III.1 Introduction

This appendix presents an example of the use of L-EQUEL for a sample academic database. The example is arranged in three sections. First, the L-EQUEL input is illustrated and discussed. Next, the output from the L-EQUEL preprocessor is presented. Finally, the output from running the program is included. For each of the sections, the information is organized by L-EQUEL command; that is, all the information pertaining to a specific L-EQUEL command is discussed together.

III.2 Background Information

The example uses a very small portion of a hypothetical academic database. The functions of the academic database might include determining where and when courses should be held, who is available to teach them, how many students are enrolled for each course, etc.

The database is named "academia", and the table used extensively in the examples is named "class". The example assumes that the "academia" database already exists, but that the "class" table does not exist.

The example executes each of the L-EQUEL commands that will be present in the initial implementation of L-EQUEL.

III.3 L-EQUEL Source Code

The source code consists of a "main" routine that calls one smaller routine for each L-EQUEL command to be included. The examples present the source code, with embedded comments.

III.3.1 leqexample

```
;       ----------
;
;       Function Name:
;           leqexample
;
;       Calling Syntax:
;           (leqexample )
;
;       Parameters:
;           none
;
;       Effects:
;           This function begins the "example L-EQUEL"
;           program. For each L-EQUEL command to be executed,
;           a separate function is called.
;           The sample database used is a hypothetical
;           "academia" database.  Within the database, a "class'
;           "class" table will be created, and data within the
;           table will be appended, manipulated, and deleted.
;
;       Returns:
;           none
;
(defun leqexample ()
    (ex_ingres "academia")        ; This initiates access to
                                  ; the "academia" database.
    (ex_create "class")           ; This creates the "class"
                                  ; table within the "academia"
                                  ; database.
    (ex_append "class")           ; This appends new tuples to
                                  ; the "class" table.
    (ex_retrieve "class")         ; Retrieve and display tuples.
    (ex_replace "class")          ; Replace the values of som
                                  ; some of the tuples.
    (ex_delete "class")           ; Delete tuples.
    (ex_destroy "class")          ; Destroy the relation.
    (ex_exit)                     ; Terminate access to the
                                  ; database.
)
```

III.3.2 ex_ingres

```
;        ----------
;
;        Function Name:
;            ex_ingres
;
;        Calling Syntax:
;            (ex_ingres dbname)
;
;        Parameters:
;            dbname - name of the database that will be
;                     "opened" for INGRES access.
;
;        Effects:
;            Enables access to the "dbname" database.
;
;        Returns:
;            none
(defun ex_ingres (dbname)
    (%% leingres dbname)
    t
)
```

III.3.3 ex_create

```
;       ----------
;
;       Function Name:
;           ex_create
;
;       Calling Syntax:
;           (ex_create tname)
;
;       Parameters:
;           tname - the name of the table to be created in
;                   the currently active INGRES database.
;
;       Effects:
;           This routine creates the "tblname" table in the
;           currently active INGRES database.
;
;       Returns:
;           none
;
;
(defun ex_create (tname)
    ( $$ lestring tname)
    ( $$ lecreate tname ( csubject c20 cnumber i4
                          clocation c20 ))
    ; to illustrate that this worked, print the table.
    ( $$ leprint tname )
    t
)
```

III.3.4 ex_append

```
;       ----------
;
;       Function Name:
;           ex_append
;
;       Calling Syntax:
;           (ex_append tblname )
;
;       Parameters:
;           tblname - name of the table to which the tuples
;                     will be appended.
;
;       Effects:
;           This routine appends a previously chosen set of
;           tuples to the "tblname" table in the currently
;           active INGRES database.
;
;       Returns:
;           none
;
(defun ex_append (tblname)
    ( %% leappend tblname  (csubject
                            clocation
    ( %% leappend tblname  (csubject
                            clocation
    ( %% leappend tblname  (csubject
                            clocation
    ( %% leappend tblname  (csubject
                            clocation
    ( %% leappend tblname  (csubject
                            clocation
    ( %% leappend tblname  (csubject
                            clocation
    ; To illustrate what just took place, the table is printed
    ( %% leprint tblname)
)
```

III.3.5 ex_retrieve

```
;       ----------
;
;       Function Name:
;           ex_retrieve
;
;       Calling Syntax:
;           (ex_retrieve tblname)
;
;       Parameters:
;           tblname - the name of the table for which values
;                     are to be retrieved.
;
;       Effects:
;           This routine retrieves specific tuples from the
;           "tblname" table of the currently active INGRES
;           database. Each tuple's fields are placed into
;           "cloc", "cs", and "cnum" and the values are
;           printed.
;
;       Returns:
;           none
;
(defun ex_retrieve (tblname)
    (setq cloc nil)
    (setq cs nil)
    (setq cnum nil)
    ($$ lestring cs)
    ($$ lestring cloc)
    ($$ leint cnum)
    ($$ lerange of "c" is tblname)
    ($$ leretrieve (cs c.csubject cloc c.clocation
                    cnum c.cnumber) where
                    c.clocation=
    ($$ leretrbgn)
    (patom "Current tuple: ")
    (patom (sprintf "%s, %s, %s" cs cnum cloc))
    (terpri)
    ($$ leretrdone)
    t
)
```

III.3.6 ex_replace

```
;        ----------
;
;        Function Name:
;            ex_replace
;
;        Calling Syntax:
;            (ex_replace tblname)
;
;        Parameters:
;            tblname - name of table in which values are
;                      to be replaced
;
;        Effects:
;            This routine replaces values in the
;            "tblname" table with new values.
;
;        Returns:
;            none
;
(defun ex_replace (tblname)
    (%% lerange of "c" is tblname)
    (%% lereplace "c" (clocation
                       c.clocation=
    (%% leprint tblname)
    t
)
```

III.3.7 ex_delete

```
;       ----------
;
;       Function Name:
;           ex_delete
;
;       Calling Syntax:
;           ( ex_delete tblname)
;
;       Parameters:
;           tblname - the table name for which tuples are
;                     to be deleted.
;
;       Effects:
;           This routine deletes a set of tuples from
;           the "tblname" table.
;
;       Returns:
;           none
;
(defun ex_delete ( tblname)
    ($$ lerange of "s" is tblname)
    ; print contents of the table before deletions
    ($$ leprint tblname)
    ($$ ledelete "s" where s.csubject=
    ; print contents of the table after deletions
    ($$ leprint tblname)
    t
)
```

III.3.8 ex_destroy

```
;        ----------
;
;        Function Name:
;            ex_destroy
;
;        Calling Syntax:
;            ( ex_destroy tblname)
;
;        Parameters:
;            tblname - the name of the table that is to be
;                      destroyed.
;
;        Effects:
;            This routine removes the table "tblname" from
;            the database.
;
;        Returns:
;            none
;
(defun ex_destroy ( tblname)
    ; print contents of the table before destroying it
    ($$ leprint tblname)
    ($$ ledestroy tblname)

    ; now the print should fail, since the table has been
    ; destroyed.
    ($$ leprint tblname)
    t
)
```

III.3.9 ex_exit

```
;       ----------
;
;       Function Name:
;           ex_exit
;
;       Calling Syntax:
;           ( ex_exit)
;
;       Parameters:
;           none
;
;       Effects:
;           This routine terminates access to the
;           currently active INGRES database.
;
;       Returns:
;           none
;
(defun ex_exit ()
    (%% leexit)
    t
)
```

III.4 Output From L-EQUEL

This section illustrates the output from running L-EQUEL on the routines of the previous section. As in the earlier section, each routine is shown separately.

The L-EQUEL preprocessor changes the spacing of the program; in the examples below, the spacing has been altered to match the L-EQUEL source programs, but the program content matches the L-EQUEL preprocessor output.

III.4.1 Leqexample

Since this source program contains no L-EQUEL source commands, the program is unchanged, except that the parameter notation of () is changed to "nil".

```
(defun leqexample nil
    (ex_ingres "academia")
    (ex_create "class")
    (ex_append "class")
    (ex_retrieve "class")
    (ex_replace  "class")
    (ex_delete  "class")
    (ex_destroy "class")
    (ex_exit)
)
```

III.4.2 ex_ingres

In this routine, the "%% leingres" s-expression was changed to the call to IIingres.

```
(defun ex_ingres (dbname )
    (IIingres (sprintf "%s" dbname) nil)
    t
)
```

III.4.3 ex_create

In this program, the "%% lecreate" and "%% leprint" were changed to be calls to IIwrite, followed by calls to IIsync.

```
(defun ex_create (tname )
    (IIwrite (sprintf "%s %s (%s)" 'create tname
        '"csubject=c20,cnumber=i4,clocation=c20"))
    (IIsync nil)

    (IIwrite (sprintf "%s %s" 'print tname))
    (IIsync nil)
    t
)
```

III.4.4 ex_append

In this routine, all occurrences of "%% leappend" are replaced with calls to IIwrite and IIsync.

```
(defun ex_append (tblname )
    (IIwrite (sprintf "%s %s (%s)" 'append tblname
    '"csubject=
         clocation=
    (IIsync nil)

    (IIwrite (sprintf "%s %s (%s)" 'append tblname
    '"csubject=
        clocation=
    (IIsync nil)

    (IIwrite (sprintf "%s %s (%s)" 'append tblname
    '"csubject=
        clocation=
    (IIsync nil)

    (IIwrite (sprintf "%s %s (%s)" 'append tblname
    '"csubject=
        clocation=
    (IIsync nil)

    (IIwrite (sprintf "%s %s (%s)" 'append tblname
    '"csubject=
        clocation=
    (IIsync nil)

    (IIwrite (sprintf "%s %s (%s)" 'append tblname
    '"csubject=
        clocation=
    (IIsync nil)

    (IIwrite (sprintf "%s %s" 'print tblname))
    (IIsync nil)
)
```

III.4.5 ex_retrieve

In this routine, the "leretrieve" information is replaced by a call to "IIwrite", followed by calls to several routines to set up the variable information and the embedded code.

```
(defun ex_retrieve (tblname )
    (setq cloc nil )
    (setq cs nil )
    (setq cnum nil )
    (IIwrite (sprintf "%s %s %s" 'range 'of
        (concat "c" "=" tblname)))(IIsync nil)
    (IIwrite (sprintf "%s (%s) %s %s" 'retrieve
        '"cs=c.csubject,cloc=c.clocation,cnum=c.cnumber"
        "where" "c.clocation=
    (IIsetup)
    (prog()
        loop
        (cond ( (IIn_get nil)
                (IIn_ret 'cs 3)
                (IIn_ret 'cloc 3)
                (IIn_ret 'cnum 6)
              )
              (t (IIflushtup 0) (return t)
              )
        )
        (cond ( (greaterp (IIerrtest) 0)
                (go loop)
              )
              ( t
                (patom "Current tuple: " )
                (patom (sprintf "%s, %s, %s"
                        cs cnum cloc ))
                (terpri )
                (go loop)
              )
        )
    )
    t
)
```

III.4.6 ex_replace

In this routine, the "%% lerange" information, the "%% lereplace" information, and the "%% leprint" information are replaced by calls to IIwrite and IIsync.

```
(defun ex_replace (tblname )
    (IIwrite (sprintf "%s %s %s" 'range 'of
        (concat "c" "=" tblname)))(IIsync nil)
    (IIwrite (sprintf "%s %s (%s) %s %s" 'replace
        "c" '"clocation=
        "c.clocation=
    )
    (IIsync nil)

    (IIwrite (sprintf "%s %s" 'print tblname))
    (IIsync nil)
    t
)
```

III.4.7 ex_delete

This routine replaces the "%% ledelete" with calls to IIwrite and IIsync.

```
(defun ex_delete (tblname )
    (IIwrite (sprintf "%s %s %s" 'range 'of
        (concat "s" "=" tblname)))
    (IIsync nil)

    (IIwrite (sprintf "%s %s" 'print tblname))
    (IIsync nil)

    (IIwrite (sprintf "%s %s %s %s" 'delete "s"
        "where" "s.csubject=
    (IIsync nil)

    (IIwrite (sprintf "%s %s" 'print tblname))
    (IIsync nil)
    t
)
```

III.4.8 ex_destroy

This routine replaces the "%% ledestroy" with calls to IIwrite and IIsync.

```
(defun ex_destroy (tblname )
    (IIwrite (sprintf "%s %s" 'print tblname))
    (IIsync nil)

    (IIwrite (sprintf "%s %s" 'destroy tblname))
    (IIsync nil)

    (IIwrite (sprintf "%s %s" 'print tblname))
    (IIsync nil)
    t
)
```

III.5 Output From Running the Program

This section illustrates the output from running the original source program. Comments are inserted to relate the output to a specific source routine.

III.5.1 ex_ingres

The "leingres" command doesn't produce any output on the screen.

III.5.2 ex_create

The ex_create routine, including the "%% leprint", causes the empty "class" table to appear, as follows:

```
class relation

|csubject            |cnumber       |clocation          |
|-------------------------------------------------------|
|-------------------------------------------------------|
```

III.5.3 ex_append

The ex_append routine, including the "%% leprint", causes the following to appear:

class relation

csubject	cnumber	clocation
Computer_Science	761	Fairchild208
Computer_Science	690	Fairchild203
Computer_Science	890	Fairchild208
Biology	670	Ackert112
Botany	840	Umberger240
English	550	Eisenhower15

III.5.4 ex_retrieve

The ex_retrieve routine causes the following to appear:

```
Current tuple: Computer Science    , 761, Fairchild208
Current tuple: Computer Science    , 690, Fairchild203
Current tuple: Computer Science    , 890, Fairchild208
```

III.5.5 ex_replace

The ex_replace routine, including the "%% leprint", causes the following to appear:

class relation

csubject	cnumber	clocation
Computer Science	761	Nichols
Computer Science	690	Nichols
Computer Science	890	Nichols
Biology	670	Ackert112
Botany	840	Umberger240
English	550	Eisenhower15

III.5.6 ex_delete

The ex_delete routine produces the following output:

class relation

csubject	cnumber	clocation
Computer Science	761	Nichols
Computer Science	690	Nichols
Computer Science	890	Nichols
Botany	840	Umberger240
English	550	Eisenhower15

III.5.7 ex_destroy

The ex_destroy routine causes the following output to appear (The INGRES error is expected, since ex_destroy tries to print the relation after it deleted it):

class relation

csubject	cnumber	clocation
Computer Science	761	Nichols
Computer Science	690	Nichols
Computer Science	890	Nichols
Botany	840	Umberger240
English	550	Eisenhower15

INGRES ERROR: 5001: PRINT: bad relation name class

III.5.8 ex_exit

The ex_exit routine produces no specific output.

APPENDIX IV - L-EQUEL Source Code

This appendix contains the working L-EQUEL source code at the time of publication of this master's report. The source code is divided into four files: l_equel.l, f_ingres.l, tbls.l, and tools.l. The code for each of the files is included.

```
;
;       File Description:
;           This file contains the high-level routines
;           to implement the L-EQUEL system.
;
;       Author:
;           Anne R. Trachsel, Summer 1985.
;
;       ----------
(declare
    (special rawfile cookfile)
    (special Stringvars Intvars Floatvars)
    (special err_nokey err_notbl err_param)
    (special err_syntax err_novar err_pqty)
    (special err_pnodecl err_retrbgn)
    (special lequeldebug)
    (special Kwrdtab)
    (*arginfo (l_equel 1 1))
    (*arginfo (parseatom 1 1))
    (*arginfo (parseexp 1 1))
    (*arginfo (parse2exp 1 1))
    (*arginfo (specproc 1 1))
)
```

```
;       ----------
;
;       Function Name:
;           (lequel_top_level)
;
;       Calling Syntax:
;           (lequel_top_level )
;
;       Parameters:
;           none
;
;       Effects:
;           This routine starts off the whole process
;           when l_equel has been "dumplisp-ed".
;
;       Returns:
;           t
;
(defun lequel_top_level ()
    (l_equel (sprintf "%s" (argv 1)) )
)
```

```
;       ----------
;
;       Function Name:
;           (l_equel)
;
;       Calling Syntax:
;           (l_equel infname)
;
;       Parameters:
;           infname -  name of file containing LISP
;                      and L-EQUEL stmts.
;
;       Effects:
;           This is the "main" routine of l_equel.
;           It calls "bgnproc" to process the file after
;           it initializes some of the global variables
;           and opens the input and output files.
;
;       Returns:
;           Always returns t.
;
(defun l_equel (infname)
    (dbg_patom "before opens") (dbg_terpri)
    ;   Setting up global error message strings:
    (init_errs)
    (setq Stringvars nil)
    (setq Intvars nil)
    (setq Floatvars nil)
    (setq rawfile (infile infname))
    (setq cookfile (outfile (substring infname 1
        (sub1 (pntlen infname)))) )
    (dbg_patom "before bgnproc") (dbg_terpri)
    (bgnproc)
    (dbg_patom "before closes")(dbg_terpri)
    (close rawfile)
    (drain cookfile)
    (close cookfile)
    (patom "L-EQUEL processing finished") (terpri)
    t
)
```

```
;
;        ----------
;
;        Function Name:
;            (bgnproc )
;
;        Calling Syntax:
;            (bgnproc)
;
;        Parameters:
;            none
;
;        Effects:
;            bgnproc processes the LISP statements that
;            are not to be interpreted by L-EQUEL. If
;            the statement is not to be processed by
;            L-EQUEL, then it is written out to the
;            output file just as it was read in.
;            If the statement is to be processed by
;            L-EQUEL, then bgnproc passes the statement,
;            including the initial "%%", to the
;            "specproc" routine.
;
;        Returns:
;            Returns t if completes successfully.
;            Returns nil otherwise.
;
(defun bgnproc ()
    (dbg_patom "Now in bgnproc") (dbg_terpri)
    (prog (next_exp)
        ; next_exp - The s_expression that is read
        ;            from the input file.
        loop
        (setq next_exp (read rawfile))
        (cond ( (null next_exp)
                (return t)
              )
        )
        (dbg_patom "send to parseexp: ")
        (dbg_patom next_exp) (dbg_terpri)
        (parseexp next_exp)
        (terpri cookfile)
        (go loop)
    )
)
```

```
;       Function Name:
;           (parseatom)
;
;       Calling Syntax:
;           (parseatom inlist)
;
;       Parameters:
;           inlist - a list that may contain atoms
;                    and lists.
;
;       Effects:
;           parseatom is a recursive routine.
;           It expects (car inlist) to be an atom.  If
;           (cadr inlist) is a list, the (cdr inlist)
;           is sent to parseexp.  If (cadr inlist) is
;           an atom, then (cdr inlist) is sent to
;           parseatom.
;
;       Returns:
;           Returns nil if (car inlist) is not an atom.
;           Returns t otherwise.
;
(defun parseatom (inlist)
    (dbg_patom "Got to parseatom")
    (dbg_terpri)
    (cond ( (not (atom (car inlist)))
            (patom
            "Errorc - parseatom expects car to be an atom")
            (terpri) nil)
          ( t (print (car inlist) cookfile)
              (patom " " cookfile ) )
    )
    (cond ( (null (cdr inlist))
            (patom ') cookfile) t )
          ( (atom (cadr inlist))
            (parseatom (cdr inlist)) t )
          ( (listp (cadr inlist) )
            (parseexp (cdr inlist)) t )
    )
    t
)
```

```
;       ----------
;
;       Function Name:
;           (parseexp)
;
;       Calling Syntax:
;           (parseexp inlist)
;
;       Parameters:
;           inlist  - the next s-expression to be
;                     parsed.
;
;       Effects:
;           parseexp is a recursive routine.
;           parseexp determines whether (car inlist)
;           is a list or an atom.  If it is an atom
;           and it is "%%", inlist is sent to
;           "specproc". If it is another atom, inlist
;           is sent to parseatom, and if (car inlist)
;           is a list, parseexp calls itself with
;           (car inlist).
;
;       Returns:
;           always returns t
;
(defun parseexp (inlist)
    (dbg_patom "Got to parseexp") (dbg_terpri)
    (cond ( (and (atom (car inlist))
                 (equal (car inlist) '%%) )
            (specproc inlist)
            t
          )
          ( (atom (car inlist))
            (patom "(" cookfile)
            (parseatom inlist) t
          )
          ( (listp (car inlist))
            (parseexp (car inlist))
            (parse2exp (cdr inlist))
          )
          ( t (patom "Errora - should not reach here")
          )
    )
    t
)
```

```
;       ----------
;
;       Function Name:
;           (parse2exp)
;
;       Calling Syntax:
;           (parse2exp inlist)
;
;       Parameters:
;           inlist  - the next s-expression to be parsed.
;
;       Effects:
;           parse2exp determines whether (car inlist) is
;           a list or an atom.  If it is an atom and it
;           is "%%", inlist is sent to "specproc". If it
;           is another atom, inlist is sent to parseatom,
;           and if (car inlist) is a list,
;           parse2exp calls parseexp with (car inlist).
;           parse2exp should only be called if the
;           car of parseexp's inlist is a list.
;
;       Returns:
;           always returns t
;
(defun parse2exp (inlist)
    (dbg_patom "Got to parse2exp") (dbg_terpri)
    (cond ( (null (car inlist))
            (patom ")" cookfile) t)
          ( (atom (car inlist))
            (parseatom inlist) t )
          ( (listp (car inlist))
            (parseexp inlist) t )
          ( t (patom "Errorb - should not reach here") )
    )
t
)
```

```
;
;       ----------
;
;       Function Name:
;           (specproc)
;
;       Calling Syntax:
;           (specproc next_exp)
;
;       Parameters:
;           next_exp - the s-expression to be processed
;                      by L-EQUEL, including the initial
;                      "%%".
;
;       Effects:
;           Determines whether the first token is a valid
;           keyword, by moving down the keyword list, one
;           keyword at a time.
;           If the token is a valid keyword, then the
;           s-expression is sent to the special-processing
;           routine that is referenced in
;           the "'op_function" property of the keyword.
;           The keywords and their properties are stored
;           in the Kwrdtab.
;           If the token is not a valid keyword, then an
;           error message is printed, and the routine
;           returns.
;
;
;       Returns:
;           If successful, returns the value of the
;           function that is called.  If not successful,
;           returns nil.
;
(defun specproc (next_exp)
    ; While still more keywords, compare car of
    ; next_exp with the car of the Keys.
    ; (car keys??)
    ; When there is a match, call the function
    ; associated with the keyword.
    (dbg_patom "Now in specproc") (dbg_terpri)
    (prog (Keys fnct)
        ; Keys - allows us to search down the Kwrdtab
        ;        list without disrupting that list.
        ; fnct - the function to execute when the
        ;        L-EQUEL function is identified.
        (setq Keys Kwrdtab)
        (dbg_patom "Cadr of next_exp is ")
        (dbg_patom (cadr next_exp))
        (dbg_terpri)
        loop
```

```
        (setq fnct nil)
        (dbg_patom "Car of keys is ")
        (dbg_patom (car Keys)) (dbg_terpri)
        (cond
            ( (null Keys)
                (patom err_nokey) (patom next_exp)
                (terpri )
                (patom err_nokey cookfile)
                (patom next_exp cookfile)
                (terpri cookfile) (return nil) )
            ( (eqstr (cadr next_exp) (car Keys))
                (dbg_patom "Going to function ")
                (dbg_patom (car Keys))
                (dbg_terpri)
                (setq fnct (get (car Keys)
                    'op_function) )
                (return (fnct (cdr next_exp))) )
            ( t (setq Keys (cdr Keys))
                (dbg_patom "Reset Keys to ")
                (dbg_patom Keys) (dbg_terpri)
                (go loop) )
        )
    )
)
```

```
;       ----------
;
;       Function Name:
;           (init_errs)
;
;       Calling Syntax:
;           (init_errs)
;
;       Parameters:
;           none
;
;       Effects:
;           This routine initializes the global variables
;           that contain the text for the l_equel
;           error messages.
;
;       Returns:
;           always returns t
;
(defun init_errs ()
    (setq err_nokey
        "Error*** Not a legal keyword: ")
    (setq err_notbl
        "Error*** No table name specified: ")
    (setq err_param
        "Error*** Invalid parameter: ")
    (setq err_syntax
        "Error*** Syntax error: ")
    (setq err_novar
        "Error*** Expected a list of variables: ")
    (setq err_pqty
        "Error*** Incomplete parameter list: ")
    (setq err_pnodecl
        "Error*** Variable parameter used but not
        declared: ")
    (setq err_retrbgn
        "Error*** Expected %% leretrbgn: ")
    t
)
```

```
;
;       File Description:
;           This file contains all the functions that
;           pertain to specific L-EQUEL commands.
;
;       Author:
;           Anne R. Trachsel, Summer 1985
;
(declare (special Stringvars Intvars Floatvars)
         (special err_nokey err_notbl err_param)
         (special err_syntax err_novar err_pqty)
         (special err_pnodecl err_retrbgn)
         (special Kwrdtab)
         (special lequeldebug)
         (special rawfile cookfile)
         (lambda f_leappend f_lecreate f_ledelete
             f_ledestroy)
         (lambda f_leexit f_leingres f_leprint
             f_lerange)
         (lambda f_lereplace f_leretrieve f_lestring)
         (lambda f_leint f_lefloat f_leretrbgn
             f_leretrdone)
         (lambda chk_crop conlist retr_wfmt)
         (*arginfo (f_leappend 1 1))
         (*arginfo (f_lecreate 1 1))
         (*arginfo (f_ledelete 1 1))
         (*arginfo (f_ledestroy 1 1))
         (*arginfo (f_ledestroy 1 1))
         (*arginfo (f_lefloat 1 1))
         (*arginfo (f_leingres 1 1))
         (*arginfo (f_leint 1 1))
         (*arginfo (f_leprint 1 1))
         (*arginfo (f_lerange 1 1))
         (*arginfo (f_leretrbgn 1 1))
         (*arginfo (f_leretrdone 1 1))
         (*arginfo (f_leretrieve 1 1))
         (*arginfo (f_lereplace 1 1))
         (*arginfo (f_lestring 1 1))
         (*arginfo (chk_crop 1 1))
         (*arginfo (conlist 1 1))
         (*arginfo (retr_wfmt 1 1))
)
```

```
;       ----------
;
;       Function Name:
;           (f_lecreate)
;
;       Calling Syntax:
;           (f_lecreate lecreate_line)
;
;       Parameters:
;           lecreate_line - the line beginning with
;                           "lecreate" - it should
;                            contain the name of the
;                            table to be created and a
;                            list of format fields for
;                            the table.
;
;       Effects:
;           Places a call to IIwrite and a call to IIsync
;           in the L-EQUEL output file.
;           The call to IIwrite contains the parameters
;           as they are passed to f_lecreate.  IIsync
;           contains a zero parameter.
;
;       Returns:
;           returns t if successful; otherwise, returns
;           nil.
;
(defun f_lecreate (create_line)
    (dbg_patom "Got to f_lecreate") (dbg_terpri)
    (prog (input_info infields var_atom tblname)
        ; input_info - used to preserve create_line
        ; infields - used to parse the list of
        ;             fields to create
        ; var_atom - formats the fields to create as
        ;             IIwrite expects them.
        ; tblname - name of table mentioned in the
        ;            create command
        (setq input_info create_line)
        (setq infields nil) (setq var_atom nil)
        (setq tblname nil)
        ;    Check for presence of "to".
        ( cond ( (equal 'to (cadr input_info))
                  (setq tblname (caddr input_info))
                  (setq input_info (cdddr input_info))
                )
                ( t (setq tblname (cadr input_info))
                    (setq input_info (cddr input_info))
                )
        )
        ;    Check for tablename
        (cond ( (null tblname)
```

```
          (print err_notbl cookfile)
          (print create_line cookfile)
          (terpri cookfile)
          (patom err_notbl) (patom create_line)
          (terpri) (return nil)
        )
        ; The next parameter exists but is not
        ; a table name.
        ( (not (atom tblname))
          (print err_notbl cookfile)
          (print create_line cookfile)
          (terpri cookfile)
          (patom err_notbl) (patom create_line)
          (terpri) (return nil)
        )
        ( t )
)

;    Now input_info should start with a list
;    having the format (name value name value ...)
(setq infields (car input_info))
(dbg_patom (list "infields:" infields))
(dbg_terpri)

(cond ( (or (null infields) (not (listp infields)))
          (print err_param cookfile)
          (print create_line cookfile)
          (terpri cookfile)
          (patom err_param) (patom create_line)
          (terpri) (return nil)
        )
)
;    Handle the first "name value" pair,
;    then to into the loop.
( cond ( (null (cadr infields))
           (patom err_pqty cookfile)
           (patom create_line cookfile)
           (terpri cookfile)
           (patom err_pqty) (patom create_line)
           (terpri)
           (return nil)
         )
         ( t
             (chk_crop (cadr infields))
             (setq var_atom
                  (concat (car infields) "="
                       (cadr infields)))
             (setq infields (cddr infields))
         )
)
loop
```

```
        (cond ( (null infields)
                ; this task is finished, so write
                ; the call to IIwrite to the output file.
                (terpri cookfile)
                (setq var_atom (sprintf "%s" var_atom))
                (print (append (list 'IIwrite
                        `(sprintf "%s %s )"
                             (quote create)
                           ,tblname (quote ,var_atom))))
                                   cookfile)
                (terpri cookfile)
                (print (list 'IIsync nil) cookfile)
                (terpri cookfile) (return t)
              )
              ( (null (cdr infields))
                (patom err_pqty cookfile)
                (patom create_line cookfile)
                (terpri cookfile)
                (patom err_pqty)
                (patom create_line) (terpri)
                (return nil)
              )
              ( t
                 ;    Create the atom that consists
                 ;    of "var=tvar,var=tvar,..."
                 (setq var_atom (concat var_atom
                     "," (car infields)
                     "=" (cadr infields)))
                 (setq infields (cddr infields))
                 (go loop)
              )
        )
    )
)
```

```
;       ----------
;
;       Function Name:
;           chk_crop
;
;       Calling Syntax:
;           (chk_crop crop)
;
;       Parameters:
;           crop - the current option to be verified
;
;       Effects:
;           This routine verifies that the field
;           definition for the current "lecreate"
;           field is a valid option.
;
;       Returns:
;           Returns t if the option is valid, and
;           returns nil otherwise.
;
( defun chk_crop (crop)
   (dbg_patom "Got to chk_crop")
   (prog (croplist numpart)
        (setq croplist (list "c" "i1" "i2" "i4"
            "f4" "f8" "date" "money"))

        (cond ( (equal (car croplist)
                    (substring crop 1 1))
                ; it is a character string field - for
                ; now no more checking is done.
                (return t)
              )
        )
        loop
        (cond ( (equal crop (car croplist))
                ; match found - we're finished.
                (return t)
              )
              ( (null croplist)
                ; no match and we're at the end
                ; of croplist. Error.
                (patom err_param cookfile )
                (patom crop cookfile)
                (terpri cookfile)
                (patom err_param )
                (patom crop ) (terpri)
              )
              ( t ; no match yet.
                (setq croplist (cdr croplist))
                (go loop)
              )
```

```
;       ----------
;
;       Function Name:
;           (f_leappend)
;
;       Calling Syntax:
;           (f_leappend append_line)
;
;       Parameters:
;           append_line - the line beginning with
;                         "leappend" - it should
;                         contain the information
;                         to be appended to the
;                         table.
;
;       Effects:
;           Places a call to IIwrite and a call to
;           IIsync in the L-EQUEL output file.
;           The call to IIwrite contains the parameters
;           as they are passed to f_leappend.  IIsync
;           contains a zero parameter.
;
;       Returns:
;           Always returns t.
;
(defun f_leappend (append_line)
    (dbg_patom "Got to f_leappend")
    (dbg_terpri)
    (prog (input_info infields var_atom tblname
            var_fmt rest_line)
        ; input_info - used to preserve append_line
        ; infields - used to parse the list of
        ;            fields to append
        ; var_atom - formats the fields to append as
        ;            IIwrite expects them.
        ; tblname - name of table mentioned in the
        ;           append command
        (setq input_info append_line)
        (setq infields nil) (setq var_atom nil)
        (setq tblname nil)
        ;    Check for presence of "to".
        ( cond ( (equal 'to (cadr input_info))
                 (setq tblname (caddr input_info))
                 (setq input_info (cdddr input_info))
               )
               ( t (setq tblname (cadr input_info))
                   (setq input_info (cddr input_info))
               )
        )
        ;    Check for tablename
        (cond ( (null tblname)
```

```
          (print err_notbl cookfile)
          (print append_line cookfile)
          (terpri cookfile)
          (patom err_notbl) (patom append_line)
          (terpri) (return nil)
        )
        ;    The next parameter exists but is
        ;    not a table name.
        ( (not (atom tblname))
          (print err_notbl cookfile)
          (print append_line cookfile)
          (terpri cookfile)
          (patom err_notbl)
          (patom append_line) (terpri)
          (return nil)
        )
        ( t )
)

;    Now input_info should start with a list having
;    the format (name value name value ...)
(setq infields (car input_info))
(dbg_patom (list "infields:" infields))

(cond ( (or (null infields) (not (listp infields)))
          (print err_param cookfile)
          (print append_line cookfile)
          (terpri cookfile)
          (patom err_param) (patom append_line)
          (terpri) (return nil)
        )
)
;    Handle the first "name value" pair,
;    then go into the loop.
( cond ( (null (cadr infields))
           (patom err_pqty cookfile)
           (patom append_line cookfile)
           (terpri cookfile)
           (patom err_pqty)
           (patom append_line) (terpri)
           (return nil)
         )
         ( t (setq var_atom
                   (concat (car infields) "="
                       (cadr infields)))
             (setq var_fmt "%s %s )")
             (setq rest_line nil)
             (setq infields (cddr infields))
         )
)
loop
```

```
(cond ( (and (null infields)
             (null (cdr input_info)))
        ; this task is finished, so write the
        ; call to IIwrite to the output file.
        (terpri cookfile)
        (setq var_atom (sprintf "%s" var_atom))
        (print (append (list 'IIwrite
                `(sprintf ,var_fmt (quote append)
                ,tblname
                  (quote ,var_atom)))) cookfile)
        (terpri cookfile)
        (print (list 'IIsync nil) cookfile)
        (terpri cookfile) (return t)
      )
      ( (null infields)
        ; We're not completely finished, but
        ; we will leave this loop.
        t
      )
      ( (null (cdr infields))
        (patom err_pqty cookfile)
        (patom append_line cookfile)
        (terpri cookfile) (patom err_pqty)
        (patom append_line) (terpri)
        (return nil)
      )
      ( t
          ;    Create the atom that consists of
          ;    "var=tvar,var=tvar,..."
          (setq var_atom (concat var_atom ","
              (car infields)
              "=" (cadr infields)))
          (setq infields (cddr infields))
          (go loop)
      )
)
(setq rest_line '())
(setq input_info (cdr input_info))
loop2
(cond ( (null input_info)
        ; Nothing more in the line, so write the
        ; proper command to the output file.
        (setq var_fmt (sprintf "%s" var_fmt))
        (setq var_atom (sprintf "%s" var_atom))
        (print (append (list 'IIwrite
                (append `(sprintf ,var_fmt
                    (quote append) ,tblname
                    (quote ,var_atom))
                    rest_line))) cookfile)
        (print (list 'IIsync nil) cookfile)
        (terpri cookfile)
```

```
              (return t)
            )
            ( t
              ; Add this atom to the list.
              (setq var_fmt (concat var_fmt " %s"))
              (setq rest_line (append rest_line
                  (list (sprintf "%s"
                      (car input_info)))))
              (setq input_info (cdr input_info))
              (go loop2)
            )
        )
    )
)
```

```
;       ----------
;
;       Function Name:
;           f_ledelete
;
;       Calling Syntax:
;           (f_ledelete delete_line)
;
;       Parameters:
;           delete_line - the line of information
;                         to be deleted.
;
;       Effects:
;           This routine places a call to IIwrite
;           and to IIsync in the output file.
;
;       Returns:
;           t
;
(defun f_ledelete (delete_line)
    (prog (dline var_fmt sline rng_var)
        (setq rng_var (cadr delete_line))
        (setq dline (cddr delete_line))
        (setq var_fmt "%s %s")
        (setq sline '())
        loop
        (cond ( (null dline)
                ; finished, so print the information
                ; to the output file.
                (setq var_fmt (sprintf "%s" var_fmt))
                (terpri cookfile)
                (print (append (list 'IIwrite
                        (append `(sprintf ,var_fmt
                            (quote delete))
                         (list rng_var) sline ))) cookfile)
                (print (list 'IIsync nil) cookfile)
                (terpri cookfile)
              )
              ( t ; still more atoms on the dline, so
                  ; add to var_fmt and to sline
                  (setq var_fmt (concat var_fmt " %s"))
                  (setq sline (append sline
                    (list (sprintf "%s" (car dline)))))
                  (setq dline (cdr dline))
                  (go loop)
              )
        )
    )
)
```

```
;        ----------
;
;        Function Name:
;            (f_leexit)
;
;        Calling Syntax:
;            (f_leexit exit_line)
;
;        Parameters:
;            exit_line - is nil.  This parameter is
;                        included so that the interface
;                         to the function routines is
;                        consistent.
;
;        Effects:
;            f_leexit places a call to the IIexit
;            routine in the L-EQUEL output file.
;            IIexit is written in C.
;
;        Returns:
;            always returns t
;
(defun f_leexit (exit_line)
    (dbg_patom "Got to f_leexit")
    (terpri cookfile)
    (print '(IIexit) cookfile) (terpri cookfile)
    t
)
```

```
;       ----------------
;
;       Funotion Name:
;           (f_leingres)
;
;       Calling Syntax:
;           (f_leingres ingres_line)
;
;       Parameters:
;           ingres_line - the line beginning with
;                         "ingres" - it should
;                         contain any INGRES options
;                         and the name of the database.
;
;       Effects:
;           Places a call to IIingres in the L-EQUEL
;           output file. The call to IIingres contains
;           the parameters as they are passed to
;           f_leingres.  A zero parameter is added
;           to the end of the list.
;
;       Returns:
;           Always returns t.
;
(defun f_leingres (ingres_line)
    (dbg_patom "Got to f_leingres") (dbg_terpri)
    (setq Stringvars '())
    (prog (nextopt ingres_ops ingops ingout fmt_var)
    ;    Setting up option string for ingres
        (setq ingres_ops (list "-u" "-c" "-i" "-f"
            "-v" "-n" "+a" "-!" "+d"
            "-d" "+s" "-s") )
        (setq ingops ingres_ops)
        (setq ingout nil)
        (setq fmt_var "")

        ; First, check whether there are any options
        ; at all. If not, we're finished.
        ( cond ( (null (cddr ingres_line))
                 (print (append (list 'IIingres
                          `(sprintf "%s"
                              ,(cadr ingres_line)) nil ) )
                          cookfile)
                 (terpri cookfile)
                 (return t)
               )
        )

        loop
        (cond ( (null (cddr ingres_line))
        ; Only thing left on command line is the database name
```

```
        ; so we can write the call to the output file.
                (setq fmt_var (sprintf "%s"
                    (concat fmt_var '" %s")))
                (setq ingout
                 (append (list 'sprintf fmt_var)
                     ingout (cadr ingres_line)))
                (print (append (list 'Ilingres
                    ingout)) cookfile)
                (terpri cookfile)
                (return t)
            )
        )
        (setq nextopt (cadr ingres_line))
        (cond ( (null ingops)    ;invalid option
                (print (cons err_param ingout)
                    cookfile)
                (terpri cookfile)
                (patom (cons err_param ingout) )
                (terpri)
                (setq ingres_line (cdr ingres_line))
                (setq ingops ingres_ops)
                (go loop)
            )
            ( (equal (substring nextopt 1 2)
                (car ingops) ) ;valid option
                (setq fmt_var (concat fmt_var '" %s"))
                (setq ingout (append ingout
                    (list nextopt)))
                (setq ingres_line (cdr ingres_line))
                (setq ingops ingres_ops)
                (go loop)
            )
            ( t (setq ingops (cdr ingops))
                (go loop) ;not found yet
            )
        )
    )
)
```

```
;       ----------
;
;       Function Name:
;           (f_leprint)
;
;       Calling Syntax:
;           (f_leprint print_line)
;
;       Parameters:
;           print_line - The arguments to the leprint
;                        command.
;
;       Effects:
;           Writes a IIwrite call to the output file.
;
;       Returns:
;           Returns t if the syntax is correct,
;           returns nil otherwise.
;
(defun f_leprint (print_line)
    (prog (printname inline var_fmt plist)
        (setq var_fmt "%s %s")
        (setq plist '())
        (setq inline print_line)
        (cond ((null (cdr inline))
               (patom err_syntax cookfile)
                   (patom print_line cookfile)
               (terpri cookfile)
               (patom err_syntax)
                   (patom print_line) (terpri)
               (return nil)
              )
        )
        ; Get past the "leprint" and set printname to
        ; the first table name.
        ; Handle the first table name separately,
        ; then go into the loop.
        (setq inline (cdr inline))
        (setq printname (car inline))

        (cond ( (null (cdr inline))
               (setq var_fmt (sprintf "%s" var_fmt))
               (terpri cookfile)
               (print (append (list 'IIwrite
                       `(sprintf ,var_fmt
                           (quote print) ,printname)))
                       cookfile)
               (terpri cookfile)
               (print (list 'IIsync nil) cookfile)
               (terpri cookfile)
               (return t)
```

```
        )
    )

    ; If we got here, there must be more than one
    ; table name in the "leprint" statement.
    loop
    (cond ( (null (cdr inline))
            (setq var_fmt (sprintf "%s" var_fmt))
            (setq plist (append plist
                (list printname)))
            (terpri cookfile)
            (print (append (list 'IIwrite
                    (append `(sprintf ,var_fmt
                        (quote print)) plist)))
                     cookfile)
            (terpri cookfile)
            (print (list 'IIsync nil) cookfile)
            (terpri cookfile)
            (return t)
          )
          ( t (setq var_fmt (concat var_fmt ",%s"))
              (setq plist (append plist
                  (list printname)))
              (setq inline (cdr inline))
              (setq printname (car inline))
              (go loop)
          )
    )
  )
)
```

```
;       ----------
;
;       Function Name:
;           f_lerange
;
;       Calling Syntax:
;           (f_lerange range_line)
;
;       Parameters:
;           range_line - the information pertaining to the
;                        lerange command.
;
;       Effects:
;           This command sends a call to IIwrite and to
;           IIsync to the output file.
;
;       Returns:
;           If no error, returns t. Otherwise, returns nil
;
(defun f_lerange (range_line)
    (prog (outline rline )
        (cond ( (not (equal (car range_line) 'lerange))
                (patom err_syntax cookfile)
                (patom range_line cookfile)
                (terpri cookfile)
                (patom err_syntax)
                (patom range_line )
                (terpri)
                (return nil)
              )
              ( (not (equal (cadr range_line) 'of))
                (patom err_syntax cookfile)
                (patom range_line cookfile)
                (terpri cookfile)
                (patom err_syntax) (patom range_line)
                (terpri)
                (return nil)
              )
        )
        (setq rline (cddr range_line))

        (cond ( (null rline)
                (terpri cookfile)
                (patom err_param cookfile)
                (patom range_line cookfile)
                (terpri cookfile)
                (patom err_param) (patom range_line)
                (terpri) (return nil)
              )
              ( (null (cddr rline))
                ; error - not enough arguments to make
```

```
                ; the next range stmt
                (patom err_param cookfile)
                (patom range_line cookfile)
                (terpri cookfile)
                (patom err_param) (patom range_line)
                (terpri) (return nil)
              )
              ( (not (equal 'is (cadr rline)))
                ; error - the middle argument of the
                ; three should be "is"
                (patom err_param cookfile)
                (patom range_line cookfile)
                (terpri cookfile)
                (patom err_param) (patom range_line)
                (terpri) (return nil)
              )
              ( t
                   (terpri cookfile)
                   (print (append (list 'IIwrite
                            `(sprintf "%s %s %s"
                                (quote range) (quote of)
                             (concat ,(car rline) "="
                             ,(caddr rline)))))
                            cookfile)
                   (print (list 'IIsync nil) cookfile)
                   (terpri cookfile)
              )
          )
      )
  )
```

```
;       ----------
;
;       Function Name:
;           f_lereplace
;
;       Calling Syntax:
;           (f_lereplace replace_line)
;
;       Parameters:
;           replace_line - the line of information that
;                          specifies the INGRES replace
;                          command.
;
;       Effects:
;           Sends calls to IIwrite and IIsync to the
;           output file.
;
;       Returns:
;           t
;
(defun f_lereplace (rplc_line)
    (dbg_patom "Got to f_lereplace") (dbg_terpri)
    (prog (input_info infields var_atom rngname
        var_fmt rest_line)
        ; input_info - used to preserve append_line
        ; infields - used to parse the list of fields
        ;            to append
        ; var_atom - formats the fields to append
        ;            as IIwrite expects them.
        ; rngname - name of range variable
        (setq input_info rplc_line)
        (setq infields nil) (setq var_atom nil)
        (setq rngname (cadr input_info))
        (setq input_info (cddr input_info))
        ;    Check rngname
        (cond ( (null rngname)
                (print err_notbl cookfile)
                (print rplc_line cookfile)
                (terpri cookfile)
                (patom err_notbl)
                (patom rplc_line) (terpri)
                (return nil)
              )
              ;    The next parameter exists but
              ;  is not a range name.
              ( (not (atom rngname))
                (print err_notbl cookfile)
                (print rplc_line cookfile)
                (terpri cookfile)
                (patom err_notbl) (patom rplc_line)
                (terpri) (return nil)
```

```
        )
        ( t )
)

;       Now input_info should start with a list
;       having the
;       format (name value name value ...)
(setq infields (car input_info))
(dbg_patom (list "infields:" infields))
(dbg_terpri)
(cond ( (or (null infields)
            (not (listp infields)))
          (print err_param cookfile)
          (print rplc_line cookfile)
          (terpri cookfile)
          (patom err_param) (patom rplc_line)
          (terpri) (return nil)
        )
)
;       Handle the first "name value" pair,
;       then to into the loop.
( cond ( (null (cadr infields))
           (patom err_pqty cookfile)
           (patom rplc_line cookfile)
           (terpri cookfile)
           (patom err_pqty)
           (patom rplc_line) (terpri)
           (return nil)
         )
         ( t (setq var_atom
                (concat (car infields)
                    "=" (cadr infields)))
             (setq infields (cddr infields))
             (setq var_fmt "%s %s )")
         )
)
loop
(cond ( (and (null infields)
             (null (cdr input_info)))
          ; this task is finished, so write the
          ; call to IIwrite to the output file.
          (terpri cookfile)
          (setq var_atom (sprintf "%s" var_atom))
          (print (append (list 'IIwrite
                   `(sprintf ,var_fmt
                       (quote replace) ,rngname
                     (quote ,var_atom)))) cookfile)
          (terpri cookfile)
          (print (list 'IIsync nil) cookfile)
          (terpri cookfile) (return t)
        )
```

```
        ( (null infields)
          ; We're not completely finished, but
          ; we will leave this loop.
          t
        )
        ( (null (cdr infields))
          (patom err_pqty cookfile)
          (patom rplc_line cookfile)
          (terpri cookfile) (patom err_pqty)
          (patom rplc_line) (terpri)
          (return nil)
        )
        ( t
            ;    Create the atom that consists of
            ;    "var=tvar,var=tvar,..."
            (setq var_atom (concat var_atom
                  "," (car infields)
                  "=" (cadr infields)))
            (setq infields (cddr infields))
            (go loop)
        )
  )
  (setq rest_line '())
  (setq input_info (cdr input_info))
  loop2
  (cond ( (null input_info)
          ; Nothing more in the line, so write the
          ; write the proper command to
          ; the output file.
          (setq var_fmt (sprintf "%s" var_fmt))
          (setq var_atom (sprintf "%s" var_atom))
          (print (append (list 'IIwrite
                  (append `(sprintf ,var_fmt
                  (quote replace) ,rngname
                   (quote ,var_atom))
                   rest_line))) cookfile)
          (print (list 'IIsync nil) cookfile)
          (terpri cookfile)
          (return t)
        )
        ( t
          ; Add this atom to the list.
          (setq var_fmt (concat var_fmt " %s"))
          (setq rest_line (append rest_line
              (list (sprintf "%s"
                  (car input_info)))))
          (setq input_info (cdr input_info))
          (go loop2)
        )
  )
)
```

)

```
;       ----------
;
;       Function Name:
;           (conlist)
;
;       Calling Syntax:
;           (conlist inparams)
;
;       Parameters:
;           inparams - a list containing input parameters
;                      to be connected into one string.
;
;       Effects:
;           This routine combines all its input parameters
;           into one string.  The atoms of the input
;           parameters are separated by commas in the
;           output string.
;
;       Returns:
;           The string formed by concatenating all the
;           input parameters, and separating them by
;           commas.
;
(defun conlist (inparams)
    (prog (nextopt outlist)
        (setq nextopt inparams) (setq outlist nil)
        (cond ( (null inparams)
                ; nothing left to do.
                (return outlist)
              )
        )
        (setq outlist (car nextopt))
        loop
        (cond ( (null (cdr nextopt))
                (return outlist)
              )
              ( t (setq outlist (concat outlist
                '|,| (cadr nextopt)))
                  (setq nextopt (cdr nextopt))
                  (go loop)
              )
        )
    )
)
```

```
;       ----------
;
;       Function Name:
;           f_lestring
;
;       Calling Syntax:
;           (f_lestring lestring_line)
;
;       Parameters:
;           lestring_line - the line for the
;                           string declaration
;
;       Effects:
;           This is the routine called when the
;           "%% lestring" function is encountered.
;           It adds the variable's name to a list of
;           variables that are declared to have
;           type "string".
;           The list is called "Stringvars".
;
;       Returns:
;           Returns "t" if it can add the variable
;           name to the list;
;           returns "nil" if the variable name is
;           invalid (i.e. not an atom).
;
(defun f_lestring (lestring_line)
    (cond ( (atom (cadr lestring_line))
            (setq Stringvars (append
                (list (cadr lestring_line)) Stringvars))
            t
          )
          ( t (patom  err_param cookfile)
              (patom lestring_line cookfile)
              (terpri cookfile)
              (patom err_param) (patom lestring_line)
              (terpri) t
          )
    )
    t
)
```

```
;        ----------
;
;        Function Name:
;            f_leint
;
;        Calling Syntax:
;            (f_leint leint_line)
;
;        Parameters:
;            leint_line - the line for the integer
;                         declaration
;
;        Effects:
;            This is the routine called when the
;            "%% leint" function is encountered.
;            It adds the variable's name to a list of
;            variables that are declared to have type
;            "integer" . The list is called "Intvars".
;
;        Returns:
;            Returns "t" if it can add the variable
;            name to the list;returns "nil" if the
;            variable name is invalid (i.e. not
;            an atom).
;
(defun f_leint (leint_line)
    (cond ( (atom (cadr leint_line))
            (setq Intvars (cons
                (cadr leint_line) Intvars))
          )
          ( t (patom  err_param cookfile)
              (patom leint_line cookfile)
              (terpri cookfile)
              (patom err_param)
              (patom leint_line) (terpri)
          )
    )
    t
)
```

```
;
;       ----------
;
;       Function Name:
;           f_lefloat
;
;       Calling Syntax:
;           (f_lefloat lefloat_line)
;
;       Parameters:
;           lefloat_line - the line for the float
;                          declaration.
;
;       Effects:
;           This is the routine called when the
;           "%% lefloat" function is encountered.
;           It adds the variable's name to a list of
;           variables that are declared to have type
;           "float". The list is called "Floatvars".
;
;       Returns:
;           Returns "t" if it can add the variable
;           name to the list;
;           returns "nil" if the variable name is
;           invalid (i.e. not an atom).
;
(defun f_lefloat (lefloat_line)
    (cond ( (atom (cadr lefloat_line))
            (setq Floatvars (cons
                (cadr lefloat_line) Floatvars) )
          )
          ( t (patom  err_param cookfile)
              (patom lefloat_line cookfile)
              (terpri cookfile)
              (patom err_param)
              (patom lefloat_line) (terpri)
          )
    )
    t
)
```

```
;       ----------
;
;       Function Name:
;           f_leretrieve
;
;       Calling Syntax:
;           (f_leretrieve retrieve_line)
;
;       Parameters:
;           retrieve_line - the s_expression associated
;                           with the "retrieve" call,
;                           including the "leretrieve".
;
;       Effects:
;           This is the top-level routine for the
;            "%% leretrieve" function.  It calls
;           several other functions that handle
;           specific pieces of the command.  The
;           end result is that the appropriate
;           calls appear in the L-EQUEL output file.
;
;       Returns:
;           Always returns t.
(defun f_leretrieve (retrieve_line)
    (prog (var_list curr_var )
        ; var_list - the list of variables used
        ;            in the command.
        ; curr_var - the current variable being
        ;             considered.
        ;            (curr_var is a member of var_list)
        ;    First, format the IIwrite command and place
        ;    it in the output file:
        (setq var_list (retr_wfmt retrieve_line))
        ;    Add the "fixed" calls to the output file:
        (print (list 'IIsetup) cookfile)
        (terpri cookfile)
        (patom "og cookfile)
        (terpri cookfile)
        (print 'loop cookfile) (terpri cookfile)
        (patom "(C)nd  n_get nil)" cookfile)
        (terpri cookfile)
        ;    Next, add the calls to IIn_ret that depend
        ;    on the variable names and types:

        loop
        (setq curr_var (car var_list))
        (cond ( (null var_list)
                t ; that is all for this section,
                  ; continue on to the next section.
              )
              ( (member curr_var Stringvars)
```

```
          (print (list 'IIn_ret (list
              'quote curr_var) 3) cookfile)
          (setq var_list (cdr var_list))
          (go loop)
        )
        ( (member curr_var intvars)
          (print (list 'IIn_ret (list
              'quote curr_var) 6) cookfile)
          (setq var_list (cdr var_list))
          (go loop)
        )
        ( (member curr_var Floatvars)
          (print (list 'IIn_ret
              (list 'quote curr_var) 2) cookfile)
          (setq var_list (cdr var_list))
          (go loop)
        )
        ( t (patom err_pnodecl cookfile)
            (patom curr_var cookfile)
            (patom retrieve_line cookfile)
            (terpri cookfile)
            (patom err_pnodecl) (patom curr_var)
            (patom retrieve_line) (terpri)
        )
    )

    ;    Next, add some more "fixed" information
    ;    to cookfile.
    (terpri cookfile)
    (patom ")" cookfile) (terpri cookfile)
    (patom " flushtup 0) turn t)"
        cookfile)
    (terpri cookfile)
    (patom ")" cookfile)
    (patom ")" cookfile)
    (patom
    "(C)nd Eaterperrtest) 0) loop)"
        cookfile)
    (patom ")" cookfile) (terpri cookfile)
    (patom " cookfile)

    ;    That is all that this routine can do.
    ;    The next %% function encountered should
    ;    be "%% leretrbgn"


  )
  t
)
```

```
;       ----------
;
;       Function Name:
;           retr_wfmt
;
;       Calling Syntax:
;           (retr_wfmt retr_line)
;
;       Parameters:
;           retr_line - the s_expression associated with
;                       the "%% leretrieve" command.  It
;                       includes the "leretrieve" but
;                       not the "%%".
;
;       Effects:
;           This routine formats the "IIwrite" call and
;           writes the call to the output file.
;
;       Returns:
;           Returns the list of variables associated with
;           the "leretrieve" command if the command is
;           successful; if there is an error, this routine
;           returns "nil".
;
(defun retr_wfmt (retr_line)
    (prog (var_list inlist var_atom var_fmt rest_line
        input_info)
        ; var_list - the list of variables used by the
        ;            retrieval (var_list is extracted
        ;            from inlist)
        ; inlist - the list of variables and their
        ;          tuple fields
        ; var_atom - the format of
        ;             var=value,var=value, etc.
        ;            expected by IIwrite.  The
        ;            var=value are extracted from inlist
        (setq var_atom nil)
        (setq var_list nil)
        (setq input_info (cdr retr_line))
        (setq inlist (car input_info))
        ;    The inlist should be a list containing
        ;    variables and their corresponding tuple field.
        (cond ( (not (listp inlist))
                (patom err_novar cookfile)
                (patom retr_line cookfile)
                (terpri cookfile)
                (patom err_novar) (patom retr_line)
                (terpri)
                (return nil)
              )
        )
```

```
(cond ( (not (null (cadr inlist)))
        (setq var_fmt "%s )")
        (setq var_atom (concat (car inlist)
            "=" (cadr inlist)))
        (setq var_list (list (car inlist)))
        (setq inlist (cddr inlist))
      )
      ( t (patom err_pqty cookfile)
          (patom retr_line cookfile)
          (terpri cookfile)
          (patom err_pqty) (patom retr_line)
              (terpri)
          (return nil)
      )
)
loop
(cond ( (and (null inlist)
             (null (cdr input_info)))
        ; this task is finished, so write the
        ; call to IIwrite to the output file.
        (terpri cookfile)
        (setq var_atom (sprintf "%s" var_atom))
        (print (append (list 'IIwrite
                `(sprintf ,var_fmt
                (quote retrieve)
                (quote ,var_atom)))) cookfile)
        (terpri cookfile)
        (terpri cookfile) (return var_list)
      )
      ( (null inlist)
        ; We're not completely finished, but we
        ; will leave this loop.
        t
      )
      ( (null (cdr inlist))
        (patom err_pqty cookfile)
        (patom retr_line cookfile)
        (terpri cookfile)
        (patom err_pqty)
        (patom retr_line) (terpri)
        (return nil)
      )
      ( t
          ;    Create the atom that consists of
          ;    "var=tvar,var=tvar,..."
          (setq var_atom (concat var_atom ","
              (car inlist) "=" (cadr inlist)))
          (setq var_list (append var_list
              (list (car inlist))))
          (setq inlist (cddr inlist))
```

```
              (go loop)
          )
      )
      (setq rest_line '())
      (setq input_info (cdr input_info))
      loop2
      (cond ( (null input_info)
              ; Nothing more in the line, so write the
              ; proper command to the output file.
              (setq var_fmt (sprintf "%s" var_fmt))
              (setq var_atom (sprintf "%s" var_atom))
              (print (append (list 'IIwrite
                      (append `(sprintf
                          ,var_fmt (quote retrieve)
                       (quote ,var_atom)) rest_line)))
                          cookfile)
              (terpri cookfile)
              (return var_list)
            )
            ( t
             ; Add this atom to the list.
             (setq var_fmt (concat var_fmt " %s"))
             (setq rest_line (append rest_line
                 (list (sprintf "%s"
                     (car input_info)))))
             (setq input_info (cdr input_info))
             (go loop2)
            )
      )
   )
)
```

```
;        ----------
;
;        Function Name:
;            f_leretrbgn
;
;        Calling Syntax:
;            (f_leretrbgn bgn_line)
;
;        Parameters:
;            bgn_line - This parameter is null; it is
;                       included for consistency with the
;                       format of calls to "%%" functions.
;
;        Effects:
;            This function currently does nothing, but is
;            included as a placeholder.  In the future, a
;            global variable should be added to L-EQUEL to
;            keep track of the last "%%" function
;            encountered. When that is implemented,
;            this function can verify that the last
;            function encountered was "%% leretrieve".
;
;        Returns:
;            Always returns t
;
(defun f_leretrbgn (bgn_line)
    t
)
```

```
;       ----------
;
;       Function Name:
;           f_leretrdone
;
;       Calling Syntax:
;           (f_leretrdone done_line)
;
;       Parameters:
;           done_line - This parameter is null; it is
;                       included for consistency with
;                       the format of %% function calls.
;
;       Effects:
;           This routine adds the last section of "fixed"
;           code for the "leretrieve" command to the
;           output file.
;
;       Returns:
;           Always returns t.
;
(defun f_leretrdone (done_line)
    ;    Add the final bit of "fixed" code for the
    ;    "%% leretreive" commmand.
    (patom " loop)" cookfile)
    (terpri cookfile)
    (patom ")" cookfile)
    (patom ")" cookfile)
    (patom ")" cookfile) (terpri cookfile)
    t
)
```

```
;       ----------
;
;       Function Name:
;           (f_ledestroy)
;
;       Calling Syntax:
;           (f_destroy dstry_line)
;
;       Parameters:
;           dstry_line - The arguments to the ledestroy
;                        command.
;
;       Effects:
;           Writes a IIwrite call to the output file,
;           followed by a IIsync call.
;
;       Returns:
;           Returns t if the syntax is correct,
;           returns nil otherwise
;
(defun f_ledestroy (dstry_line)
    (prog (dstryname inline var_fmt plist)
        (setq var_fmt "%s %s")
        (setq plist '())
        (setq inline dstry_line)
        (cond ((null (cdr inline))
                (patom err_syntax cookfile)
                (patom dstry_line cookfile)
                (terpri cookfile)
                (patom err_syntax)
                (patom dstry_line) (terpri)
                (return nil)
              )
        )
        ; Get past the "ledestroy" and set dstryname to
        ; the first table name.
        ; Handle the first table name separately, then
        ; go into the loop.
        (setq inline (cdr inline))
        (setq dstryname (car inline))

        (cond ( (null (cdr inline))
                (setq var_fmt (sprintf "%s" var_fmt))
                (terpri cookfile)
                (print (append (list 'IIwrite
                        `(sprintf ,var_fmt
                            (quote destroy) ,dstryname))
                        cookfile)
                (terpri cookfile)
                (print (list 'IIsync nil) cookfile)
                (terpri cookfile)
```

```
                (return t)
            )
        )

        ; If we got here, there must be more than
        ; one table name in the "ledstroy" statement.
        loop
        (cond ( (null (cdr inline))
                (setq var_fmt (sprintf "%s" var_fmt))
                (setq plist (append plist
                    (list dstryname)))
                (terpri cookfile)
                (print (append (list 'IIwrite
                        (append `(sprintf ,var_fmt
                            (quote print)) plist)))
                         cookfile)
                (terpri cookfile)
                (print (list 'IIsync nil) cookfile)
                (terpri cookfile)
                (return t)
            )
            ( t (setq var_fmt (concat var_fmt ",%s"))
                (setq plist (append plist
                    (list dstryname)))
                (setq inline (cdr inline))
                (setq dstryname (car inline))
                (go loop)
            )
        )
    )
)
```

```
;
;       File Description:
;           This file contains the routines that are called
;           to load information into the L-EQUEL keyword
;           table and operator tables.
;
;       Author:
;           Anne R. Trachsel, Summer 1985.
;
;       ---------
(declare (lambda add_kwrd add_op)
        (special Kwrdtab Optab)
        (*arginfo (add_kwrd 2 2))
)
```

```
;        ---------
;        Function Name:
;            add_kwrd
;
;        Calling Syntax:
;            (add_kwrd op_name op_function)
;
;        Parameters:
;                op_name     - name of the token
;                op_function - the function to be
;                              called when this token
;                              is encountered.
;
;        Effects:
;            This routine sets the op_function properties
;            of the token known as op_name.  It then adds
;            the op_name token to the Kwrdtab list.
;
;        Returns:
;            the value of op_name
;
(defun add_kwrd (op_name op_function)
    (putprop op_name op_function 'op_function)
    (setq Kwrdtab (cons op_name Kwrdtab))
    op_name
)
```

```
;
;       File Description:
;           This file contains tools used by
;           the L-EQUEL preprocessor.
;
;       Author:
;           Anne R. Trachsel, Summer 1985
;
;       ----------
(declare (lambda dbg_patom dbg_terpri kw_populate)
         (lambda add_kwrd add_op)
         (lambda loadit set_debug)
         (special Kwrdtab Optab)
         (special lequeldebug)
         (*arginfo (add_kwrd 2 2))
         (*arginfo (dbg_patom 1 1))
         (*arginfo (set_debug 1 1))
)
```

```
;       ----------
;
;       Function Name:
;           (dbg_patom)
;
;       Calling Syntax:
;           (dbg_patom dbg_stmt)
;
;       Parameters:
;           dbg_stmt - the debugging information
;                      to be printed (patom'd)
;
;       Effects:
;           If the global variable "lequeldebug" is
;           set to zero, then this routine is null.
;           If "lequeldebug" is set to one, then
;           this routine patom's its argument.
;           This allows debugging print statements
;           to remain in completed code without
;           affecting normal running of the program.
;
;       Returns:
;           always returns t
;
(defun dbg_patom (dbg_stmt)
    (cond  ( (zerop lequeldebug) t)
           ( (onep lequeldebug) (patom dbg_stmt) t)

    )
)
```

```
;         ----------
;
;         Function Name:
;             (dbg_terpri)
;
;         Calling Syntax:
;             (dbg_terpri)
;
;         Parameters:
;             None.
;
;         Effects:
;             If the global variable "lequeldebug" is
;             set to zero, then this routine is null.
;             If "lequeldebug" is set to one, then
;             this routine performs a "terpri".
;             This allows debugging print statements to
;             remain in completed code without affecting
;             normal running of the program.
;
;         Returns:
;             always returns t
;
(defun dbg_terpri ()
    (cond  ( (zerop lequeldebug) t)
           ( (onep lequeldebug) (terpri) t)

    )
)
```

```
;       ---------------
;
;       Function Name:
;           (kw_populate)
;
;       Calling Syntax:
;           (kw_populate)
;
;       Parameters:
;           none
;
;       Effects:
;           Populates the Kwrdtab table for testing.
;           At the beginning of the routine, Kwrdtab
;           is set to nil. For this routine to work,
;           the file tbls.l must have already been
;           loaded in, since this uses the routine
;           "add_kwrd".
;
;       Returns:
;           Always returns true.
;
(defun kw_populate ()
    (setq Kwrdtab nil)
    (add_kwrd 'leappend 'f_leappend)
    (add_kwrd 'lecreate 'f_lecreate)
    (add_kwrd 'ledelete 'f_ledelete)
    (add_kwrd 'ledestroy 'f_ledestroy)
    (add_kwrd 'leexit 'f_leexit)
    (add_kwrd 'lefloat 'f_lefloat)
    (add_kwrd 'leingres 'f_leingres)
    (add_kwrd 'leint 'f_leint)
    (add_kwrd 'leprint 'f_leprint)
    (add_kwrd 'lerange 'f_lerange)
    (add_kwrd 'lereplace 'f_lereplace)
    (add_kwrd 'leretrieve 'f_leretrieve)
    (add_kwrd 'leretrbgn 'f_leretrbgn)
    (add_kwrd 'leretrdone 'f_leretrdone)
    (add_kwrd 'lestring 'f_lestring)
    t
)
```

```
;       ----------
;
;       Function Name:
;           (loadit)
;
;       Calling Syntax:
;           (loadit)
;
;       Parameters:
;           none
;
;       Effects:
;           loads in the LISP files necessary
;           to run l_equel, and populates the Kwrdtab.
;           Also initializes the value of the
;           debugging variable "lequeldebug".
;
;       Returns:
;           t
;
(defun loadit ()
    (load 'f_ingres) (load 'l_equel)
    (load 'tbls) (kw_populate)
    (set_debug 0)
    (cfasl 'IIingres.o '_IIingres 'IIingres
        "integer-function" "-lq")
    (cfasl 'IIwrite.o' _IIwrite 'IIwrite
        "integer-function" "-lq" )
    (cfasl 'IIcvar.o '_IIcvar 'IIcvar
        "integer-function" "-lq" )
    (cfasl 'IIexit.o '_IIexit 'IIexit
        "integer-function" "-lq" )
    (cfasl 'IIflushtup.o '_IIflushtup 'IIflushtup
        "integer-function" "-lq" )
    (cfasl 'IIgettup.o '_IIgettup 'IIgettup
        "integer-function" "-lq" )
    (cfasl 'IIn_get.o '_IIn_get 'IIn_get
        "integer-function" "-lq" )
    (cfasl 'IIn_ret.o '_IIn_ret 'IIn_ret
        "integer-function" "-lq" )
    (cfasl 'IIsetup.o '_IIsetup 'IIsetup
        "integer-function" "-lq" )
    (cfasl 'IIsync.o '_IIsync 'IIsync
        "integer-function" "-lq" )
    (cfasl 'IIw_left.o '_IIw_left 'IIw_left
        "integer-function" "-lq" )
    (cfasl 'IIw_right.o '_IIw_right 'IIw_right
        "integer-function" "-lq" )
    (getaddress '_IIcvar 'IIcvar "integer-function")
    (getaddress '_IIerrtest 'IIerrtest
        "integer-function")
```

```
    (getaddress '_IIexit 'IIexit "integer-function")
    (getaddress '_IIflushtup 'IIflushtup
        "integer-function")
    (getaddress '_IIgettup 'IIgettup
        "integer-function")
    (getaddress '_IIingres 'IIingres
        "integer-function")
    (getaddress '_IIn_get 'IIn_get "integer-function")
    (getaddress '_IIn_ret 'IIn_ret "integer-function")
    (getaddress '_IIsetup 'IIsetup "integer-function")
    (getaddress '_IIsync 'IIsync "integer-function")
    (getaddress '_IIw_left 'IIw_left "integer-function")
    (getaddress '_IIw_right 'IIw_right
        "integer-function")
    (getaddress '_IIwrite 'IIwrite "integer-function")
)
```

```
;       ----------
;
;       Funotion Name:
;           (set_debug)
;
;       Calling Syntax:
;           (set_debug debug_flag)
;
;       Parameters:
;           debug_flag - if it equals zero, the
;                        debugging stmts are turned
;                        off; if it equals one, the
;                        stmts are turned on.
;
;       Effects:
;           Sets or clears the debugging variable,
;           "lequeldebug".
;           This variable controls whether or not
;           debugging statements are printed.
;
;       Returns:
;           Returns t if lequeldebug can be set to a
;           valid value; returns nil otherwise.
;
(defun set_debug (debug_flag)
    (cond ( (or (zerop debug_flag)
                (onep debug_flag) )
            (setq lequeldebug debug_flag) t
          )
          ( t  (patom
                "Invalid value for lequeldebug0)
          (terpri)
          )
    )
)
```

L-EQUEL: AN EMBEDDED QUERY LANGUAGE
FOR FRANZ LISP

by

ANNE ROBERTA TRACHSEL

B.S., The Ohio State University, 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

ABSTRACT

This master's report describes the design and implementation of the LISP-Embedded Query Language, L-EQUEL. EQUEL, Embedded Query Language, enables the C language programmer to embed INGRES queries within a C language program. L-EQUEL, a set of EQUEL-based database access routines for LISP, enables the Franz LISP programmer to embed INGRES database queries within a Franz LISP program.

Current LISP databases depend on features unique to LISP; the information stored in them is not accessible to programs written in other languages, because the databases are enclosed within the program's storage area. By allowing access to the INGRES database from within a LISP program, L-EQUEL provides a facility for data sharing between programs written in Franz LISP and programs written in other languages.