FLEXIBLE ENVIRONMENTS IN DYNAMIC LEXICAL ANALYSIS SYSTEMS

by

Matthew G. Denman

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Dr. H. R. Hartson

Dr. T. Lindquist

Dr. G. W. Gorsline

May , 1984
Blacksburg, Virginia

ABSTRACT

FLEXIBLE ENVIRONMENTS IN DYNAMIC LEXICAL ANALYSIS SYSTEMS

by

Matthew G. Denman

In this thesis, a system for studying human/computer interfaces is introduced. The human/computer interface provides several features, the most noteable of which is TOKEN COMPLETION. These features permit the user to define and/or redefine command tokens, define and/or delete synonym and noiseword tokens, and to establish a terminal environment. The terminal environment includes the ability to specify automatic comment blocking, token look-ahead, and to control the source of data input (keyboard, VMS file, or I/O buffer).

The ability to token complete is based on a forest of generalized trees used to implement dynamic deterministic finite state automata (DDFA). These trees are built during IPL and loaded with command, synonym, and noiseword tokens, all of which are stored in separate VMS files. Synonym and noiseword translation is carried out in the lexical analysis process, thereby negating any need to specify these functions in the grammar of the language. Insertion and deletion into the forest may be executed at any time, permitting the dynamic definition and

deletion of synonyms and noisewords. During synonym and/or noiseword definition, lexical analysis switches to a deterministic finite state automata (DFA) mode of operation. Upon completion, lexical analysis reverts to DDFA mode.

A sample grammar is provided in the APPENDICES. The lexical analysis process is not tied into this grammar but rather is very general and will process any tokens stored in the command, synonym, and noiseword files. The sample grammar is LALR(1).

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. H. R. Hartson who believed me when I said, "I'll have the job done in six months"...and continued to believe me for the next two years. Thanks are also due to Dr. Tim Lindquist and Dr. George Gorsline who assisted me as members of my advisory committee. I would also like to thank Jon Simasek who contributed his blood and hours, days, and weeks of his time to aid in the design and implementation of the software and spent many hours reading my thesis to insure its "semantic integrity". Thanks also goes to Sandy Bachman, who provided a place to stay while I vacationed in Blacksburg from my job to work on my thesis, and again to Jon Simasek and to Janet Politis both of whom aided me in the design and drafting of the figures. Lastly, I give thanks to the Central Intelligence Agency, who provided me with the time to travel to Blacksburg.

I dedicate this thesis to my family, especially my sister who received her Bachelor's Degree, continued her education, received a Master's Degree, and is now a Ph.D. candidate...all while I was struggling to obtain my Master's Degree. Sibling rivalry forced me to continue my studies until I too received my Master's Degree.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

CHAPTER I

1. INTRODUCTION

The Computer Science Department at Virginia Tech has a student-implemented relational database management system called the Mini-Database (MDB). The MDB is used for teaching as well as for a few modest database applications. In contrast to a commercial database management system for which stability of characteristics is a prime requirement, the MDB has a continuing need to change its characteristics to meet the ever-changing requirements of a research environment. Every year features are added, removed, and changed; the user interface and query language are subject to continual and extensive alterations, primarily by the students of two graduate courses on database systems. The first course is an introduction to database theory, models, and languages, which uses the MDB as an instructional vehicle to introduce the students as users to a small relational system. The second course is a design and construction course that uses the MDB as a laboratory tool providing a test bed for various approaches to designing the parts of a DBMS.

Another graduate course, on human-computer interaction and the application of human factors research results to human-computer interfaces, also uses the MDB for teaching and for laboratory work in the design of interaction sequences, comparison of command language constructs, naming of parameters, and for testing the effectiveness of system error messages. For these purposes, the MDB interface must be

very flexible, including the ability to support the rapid modification of the query language.

Designing, redesigning, and extending the MDB has been going on in an ad hoc, manual way for several years. This paper reports on some tools and techniques utilized to streamline these processes.

CHAPTER II

2. SYSTEM OVERVIEW

The system architecture of the MDB is diagrammed schematically in Figure 1. This paper describes the use of generalized software to provide flexibility in the lexical analysis operations. Each of the parts of Figure 1 is described here briefly. The process of lexical analysis is then detailed.

2.1 QUERY LANGUAGE

The query language is an extended version of SEQUEL (SQL). A full compliment of control structures normally found in high level programming languages are defined, including DO...WHILE, IF...THEN...ELSE, DO...UNTIL, a case statement, etc. In addition, there are a wide assortment of built-in functions such as PL/I-like string functions, statistical functions, etc. Figure 2 shows a sample program. Note that the program can be entered for later compilation or interpreted on a command by command basis. For a complete overview of the BNF grammar for the query language, refer to Appendix A.

2.2 LEXICAL ANALYSIS

Lexical analysis is achieved through the use of a dynamic deterministic finite state automata (DDFA). This concept is not entirely new and was referred to as an "extended deterministic automata" by [LEWIS78]. The most prominent and unique use of this feature is token completion in which tokens may be partially or fully completed automatically when the lexical environment enters a state of non-ambiguity. This process

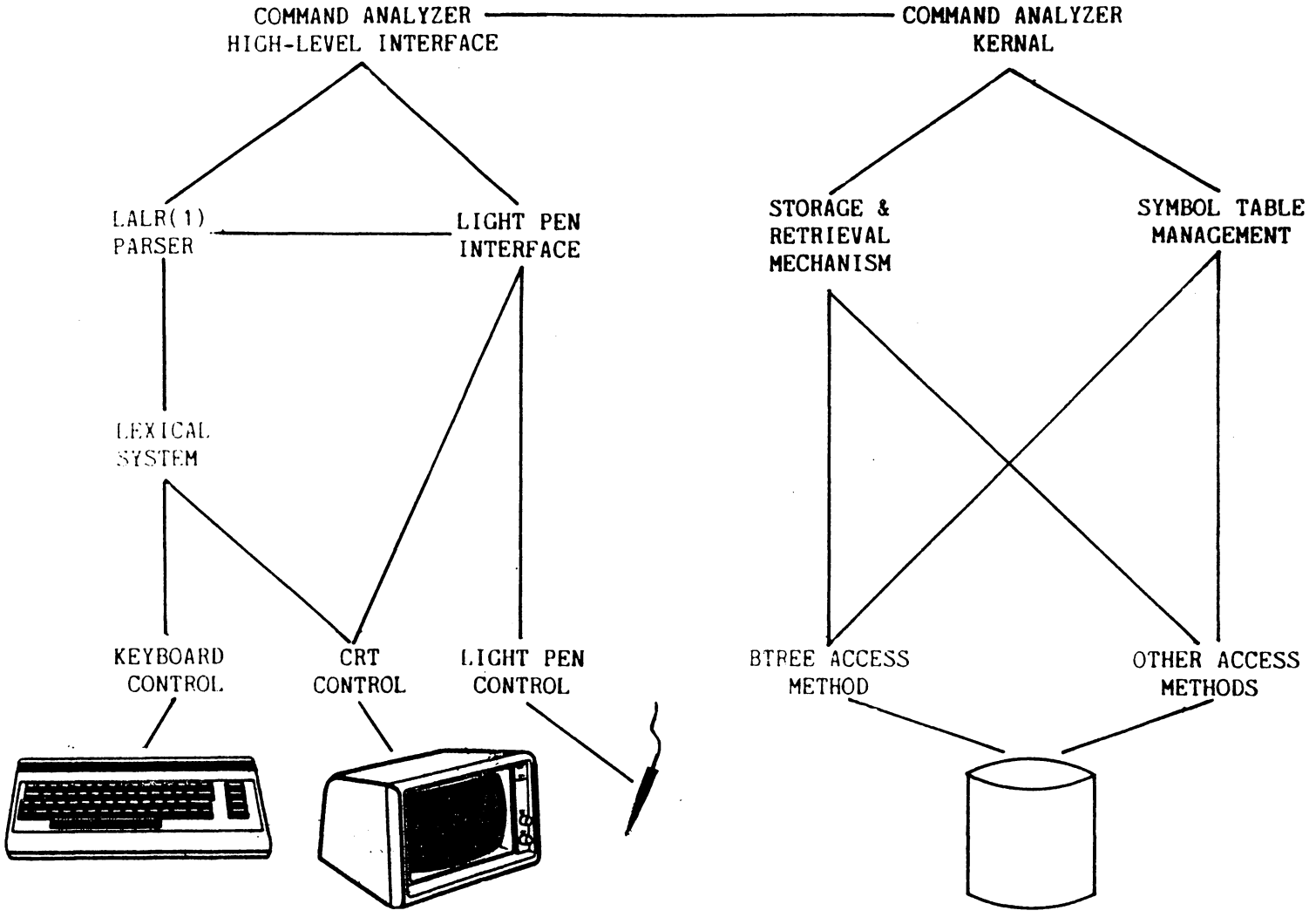


Figure 1: Overview of the Database Management System

RELATION: Sales

INVOICE_NO	AMOUNT_OF_SALE	EMPLOYEE_NO

RELATION: Employees

EMPLOYEE_NO	NAME	ADDRESS	OFFICE	SALARY

QUERY: Cut bonus checks of 10% of salary for all salesmen in Virginia who have sold $200,000 to date.

PROGRAM:

```
/* Create a new relation named, employee_no_table, that      */
/* consists of unique employee numbers in ascending order    */
SELECT employee_no UNIQUE FROM  sales ASCENDING ON employee_no;
ALIAS employee_no_table CURRENT_RELATION;
/* Loop for each unique employee number                      */
DO times<==1,COUNT(employee_no_table)
  /* For each employee number, create a relation that        */
  /* consists of the amount of sales for that employee and   */
  /* the location of his/her office                          */
  SELECT amount_of_sale FROM sales WHERE employee.no IN
    SELECT office FROM employees WHERE
      employee_no=employee_no_table(times) AND;
      office='Virginia';
  /* If the current relation created by the previous         */
  /* select for the current employee number is not empty,    */
  /* then if the total amount of sales for this employee     */
  /* number is at least 200,000 dollars, then create a       */
  /* relation that consists of the employee number's         */
  /* name, address, and salary and call a routine named      */
  /* cut_check that will cut a bonus check using this        */
  /* relation for the employee                               */
  IF COUNT(RELATION,CURRENT_RELATION)<>0
    THEN
      REAL total_sales<==SUM(ATTRIBUTE,
                         CURRENT_RELATION.amount_of_sales);
        IF total_sales>=200,000
          THEN
            SELECT name,address,salary FROM employees WHERE
              employee_no=emp_no_table(times);
            RUN out_bonus_check;
        ENDIF;
  ENDIF;
ENDIF;
```

Figure 2: Sample Query

is a function of the number of characters of a token entered at a given point in time by the user and the number of tokens beginning with that prefix. For example, if only one token begins with that prefix, then the token may be completed automatically by the lexical analysis process. Other features include token deletion, automatic line management, comment blocking, and the support of synonyms and noisewords. The lexical analysis processes are the most significant piece of work in this system.

2.3 PARSING

The parsing mechanism is LALR (look-ahead, left-right-scan, rightmost derivation) and was developed and is modified with the aid of a Bell Labs parser generator called YACC (Yet Another Compiler-Compiler) and two FORTRAN software packages. The first software package was written as an interface between the MDB and YACC. The grammar is stored as a relation in the MDB itself to facilitate modification. This interface converts the grammar relation to a form acceptable by YACC. YACC, in turn, uses the grammar to generate an LALR parser in pseudocode. The second software package was written to convert the YACC-produced pseudocode parser into FORTRAN. It is interesting to note that the second software package is itself an LALR parser generated by YACC and hand-translated to FORTRAN. An important feature is that clauses comprising a command line may appear in any order. For example, the user may type "DELETE...FROM...WHERE..." or "DELETE...WHERE...FROM..." and still convey the same meaning. This property is called clause order independence and frees the user from having to memorize any

fixed ordering of clauses within a command. Furthermore, the structure of a given clause receives the same treatment regardless of where the clause appears in any production. For example, to reference relations from the query language the clause "RELATION <relation name>" is used, regardless of where the relation name appears in any construct. This concept and others, such as semantic and syntactic complexity, concerning the design of "easy to use" languages is treated in depth in [REIS78].

To provide clause order independence, a list was produced for each command containing all possible permutations of keywords that could follow. Any entry in the list that was not semantically valid was eliminated. An example is the pairing of nested "WHERE" clauses with the appropriate "SELECT" command of a nested "SELECT". A stricter syntax is used that does not permit complete clause order independence in order to insure that the proper "WHERE" clause is matched to the proper "SELECT" in the nested "SELECT" command. Therefore, all but one of the permutations of the "WHERE" clause in the "SELECT" command was eliminated. An analogy is the "IF...THEN...ELSE" ambiguity problem in parsing procedural languages which is resolved by adopting the rule that the most recent "ELSE" is paired with the most recent unpaired "IF...THEN" [AHO79]. In the query language, the "WHERE" clause is matched to the most recent "SELECT" and the entire nested "SELECT" is terminated by a semicolon. This construct is unique in that the semicolon is required as a statement terminator. In all other statements, the semicolon is present only to provide syntactic

consistency; in otherwords, rather than being the exception for the nested "SELECT", the use of a semicolon at the end of a statement is a rule for all statements. The parser software is not implemented.

2.4 COMMAND ANALYSIS

The command analyzer is a group of modules that directs execution to the proper semantic routines defining the various commands. The command analyzer consists of a set of primitives and a kernel for executing these primitives. Interfaces, such as those for light pens, compilers, interpreters, etc., execute calls to the set of primitives which are then executed by the kernel directing processing to the appropriate command routines. For example, if a query is to be interpreted rather than compiled, the lexical analyzer processes the input and passes appropriate information to the parser. When the parser has received sufficient input to identify a complete query (usually by means of an end_of_statement delimiter), a call is placed to the command analyzer which, after determining which interface was specified, places a call to the interpreter interface which in turn makes calls to the set of primitives (PCode-like mechanism). These primitives are executed by the kernel and appropriate storage and retrieval mechanisms are called. If compilation is specified, then the compiler interface is called and object code is generated for later execution. The important point is that the storage and retrieval mechanism has been separated from the command analysis process facilitating the implementation of additional interfaces (e.g., lightpen) via the set of primitives. This software is not implemented.

2.5 STORAGE AND RETRIEVAL

The storage and retrieval mechanisms have remained as described in the MDB User's and System Description Manual [VATECH80]. The data storage and retrieval mechanism might be modified with less effort than in the current MDB, since these operations have now been separated from the command analysis procedures.

2.6 THE LEXICAL SYSTEM VERSUS LEXICAL ANALYSIS

The lexical system is that piece of software that is comprised of a set of modules for performing not only lexical analysis, but in addition, token completion, token deletion, comment blocking, I/O, token look-ahead, noiseword and synonym processing, and any additional features discussed later on. Lexical analysis is the function performed by that portion of the lexical system involved in the recognition of valid tokens. This concept is referred to as word identification by [LEWIS78]. The lexical system is implemented.

CHAPTER III

3. OVERVIEW OF THE LEXICAL SYSTEM

The following sections present a brief overview of the lexical system. System features and implementation methods are only touched upon here; a detailed description is presented in a later section.

3.1 FEATURES

In order to provide a flexible engineering and research environment, several features had to be addressed. Most of these features were eventually implemented within the lexical system and are described here. These include the ability to:

1. change a command name without changing its functions,
2. add or delete synonyms for command names,
3. add or delete noisewords
4. alter error and warning messages,
5. change system constant values without affecting the operations of the DBMS or requiring any coding modification, and
6. provide a development system, the purpose of which is to aid in the management of lexical software modification.

The first feature permits the study of effects of command names on a user. For example, a researcher might wish to determine whether there is a difference in user performance when the command "SELECT" is changed to a functionally equivalent command "CHOOSE". In order to perform this type of investigation, there must be a facility that allows the researcher to change the system's recognition of one command name to another functionally equivalent command name.

The second feature permits the study of synonym usage. The researcher may not want to alter the original command name set, but may instead want to define synonyms of various commands to determine if these synonyms facilitate the use of the language, or perhaps to determine if these synonyms cause interference, hindering the learning of the language. A facility must exist that allows the researcher to quickly define a new set of synonyms to be recognized for a given command name. A study might also be made to determine how users will most likely misspell command names and to incorporate these misspellings as synonyms. In addition, foreign languages might be supported by defining the foreign language equivalent of each command as a synonym. The differences between these first two features is somewhat cloudy given that the issues related to the first feature can be easily studied utilizing the second feature by defining synonyms rather than changing commands.

The third feature permits the study of flexible query languages by allowing the researcher to specify that certain words are meaningless and are to be ignored. Such words as "THE", "THOSE", "IN", "TO", etc. could fall into this category and are called noisewords. This facility would allow a user to enter "SELECT THE", "SELECT THOSE", or just "SELECT" since "THE" and "THOSE" are discarded.

The fourth feature permits a researcher to alter the appearance of error and warning messages in an attempt to determine the form of the message that will be most easily understood by the user. The facility simply needs to know which error message to change and what is the new error

message. The code is altered automatically. This feature is not currently implemented.

The fifth feature helps the researcher to transport the system from one machine to another (barring differences in FORTRAN and I/O), or to completely redefine system characteristics from one session to the next. Such constants as backspace, line feed, carriage return, screen size, etc. can be easily altered by changing the value of the corresponding constant variable.

The sixth feature reduces the effort required to manage the modification and development of software. There are provisions for managing libraries, source code modules, logs, etc. For example, one particular facility maintains a log of all source code modules edited but not recompiled. When the next DBMS session begins, the log is inspected and each modified module is recompiled and relinked to the system. These facilities may eliminate the drudgery of remembering whether a module was recompiled, whether the updated module was stored back in the proper library, etc. All of these functions are automated. This feature is partially implemented.

In addition, there are features which address the end user's perspective -- the operational environment of a target DBMS. Users can define their own local command names, synonyms (including misspellings), noisewords, and error and warning messages, providing for a flexible query language and environment tailored to the user's needs.

In both the research and operational environments, an attempt was made to allow as much flexibility as possible. If such flexibility is not required, the system relies on system default values.

3.2 SOFTWARE/HARDWARE

The software resides on a VAX 11/780 and is coded in VAX FORTRAN. The modules that are responsible for implementing the features enumerated earlier comprise the lexical system. Some of the modules, however, also serve as support for the LALR parser, command analyzer, and command routines.

The code takes full advantage of the features of VAX FORTRAN which is a superset of ANSI standard FORTRAN 77. No GOTO's or CONTINUE's are used. Complete indentation prevails, following a set of standardized rules. Variable names take full advantage of the 32 character limit to fully spell out mnemonic names.

A set of PL/I-like string functions is implemented. Such functions as "concatenation" operate as they would in PL/I, not as they otherwise would in VAX FORTRAN. The syntax of the functions is similar to those in PL/I. PL/I syntax was chosen since the constructs are more English oriented than they are in FORTRAN; that is, FORTRAN is operator oriented whereas PL/I provides functions for those FORTRAN operators. In addition, FORTRAN does not handle varying length strings. Therefore there was a requirement to write FORTRAN string handling functions that were capable of manipulating varying length strings. There is also a set of stack functions. These two sets comprise the support routines mentioned above and are themselves coded in FORTRAN.

There exist numerous constant values, many of which are non-printable characters. To aid the programmer with the task of code modification or development, these non-printable constants are given mnemonic names. For example, the carriage return character is referenced by the variable "carriage_return". These constants are all declared and placed in common blocks in a single "include" file, and are all initialized in a second "include" file. This greatly simplifies coding since any reference to a constant within a subroutine is resolved by simply including the first "include" file at the top of the subroutine. To initialize all constants, the second "include" file is included in the top level routine.

Variables that are referenced system wide are also declared in an "include" file. These files minimize the number of parameters passed in subroutine calls and helps to minimize the number of variable names by ensuring that a variable serves one and only one function and that its name does not change anywhere throughout the system. Even in the cases where parameter passing is required, 99% of the variable names do not change from one subroutine to the next and no variables ever change their function. The other 1% are passed to extremely generalized subroutines which are called from many different points from within the system, such as the PL/I-like functions, necessitating the name changes. There are additional "include" files, each with a specific purpose, a few of which follow:

1. PL/I-like functions,
2. stack functions,
3. system environmental control parameters,

4. DFA (Deterministic Finite Automata) related data structures, and
5. DDFA (Dynamic DFA) related data structures.

I/O is provided by calls to system service routines and is performed on a character-by-character basis. That is, as each key is depressed, the corresponding character is received, processed, and possibly echoed back to the terminal.

3.3 A FLEXIBLE QUERY LANGUAGE

The lexical system supports the definition of a flexible query language. This facility permits the user to redefine the lexical environment to more closely match his/her prose style. The following is a description of the component parts of this facility.

3.3.1 TOKEN TYPES

The lexical analyzer analyzes all tokens, of which there are four types:

1. commands (keywords such as SELECT, FROM, etc.),
2. user defined (relation, attribute, and variable names; literals, etc.),
3. synonyms (refer back to 3.1), and
3. noisewords (refer back to 3.1).

The ability to handle these last two types of tokens was greatly simplified by dealing with them in the lexical process negating the need to include them in the formal specification of the grammar for the LALR parser. The result is a smaller set of meaningful constructs requiring recognition by the parser. Thus, less software is required to implement the grammar rules via the LALR parser. The actual process of recognizing and discarding noiseword tokens is referred to as filtering

and is implemented with a filter. The filter is a small interface between the lexical system and the parser. If the filter determines that a token is a noiseword, a return is made to the lexical system to obtain the next token; otherwise the token information is passed to the parser via a call. The process of noiseword and synonym token recognition is described in the next chapter.

3.3.2 FILE TYPES

There are three files:

1. a command file,
2. a synonym file, and
3. a noiseword file.

The command file contains a sequential listing of all command tokens. At the start of each DBMS session, this file is the first to be accessed and each command token is sequentially loaded into the system. At the same time, each command token is assigned the unique token value that is stored with the token in the file. These token values, rather than the tokens themselves, are passed back to the parser. In other words, the token value is, to the parser, a specification of the attributes of that token.

The synonym file is the second file to be accessed at the start of a DBMS session. An entry in this file contains a command token followed by any number of synonym tokens, followed by a blank, another command token, a set of synonym tokens, etc. The command token is mapped into the system in order to acquire its token value (remember that command tokens have already been loaded). Next, each synonym token that follows

the command token in the synonym file is loaded and assigned the token value of the command token. Thus, a command token and its synonym(s) are assigned equivalent attributes.

The last file to be accessed at the start of a DBMS session is the noiseword file. This file contains a list of tokens to be recognized but ignored by the lexical analyzer. Recognition is based on a unique token value assigned to the noiseword.

Note that the noiseword file is used to explicitly define noisewords. An alternative is to eliminate this file and to "assume" that any unrecognizable token is a noiseword. This has the disadvantage of not permitting the detection of error conditions since all erroneous inputs are now interpreted as noisewords. This situation could complicate the learning of the query language due to a lack of negative feedback (via the error messages). However, if the elimination of the noiseword file was implemented as a user option for the experienced user, then the processing of tokens would be faster since the software to determine if a token is a noiseword and to detect and process error conditions could now be by-passed. Thus, a novice user would retain the negative feedback while learning the query language and have to explicitly define noisewords whereas the experienced user could dispense with defining noisewords and dealing with error messages by explicitly defining "non-comprehension" as a "noiseword condition". This concept has been implemented in "computer games" for home microcomputers by [SCREEN83,CARROL82].

The user defined tokens are loaded into the lexical system as they are defined in a DBMS session. At the start of each DBMS session, these tokens are acquired from the DBMS system tables and loaded. For example, relation names are acquired from the system directory and the attribute names are acquired from the relation directories.

3.4 THE HUMAN/COMPUTER INTERFACE

The human/computer interface deals with communications between the user or programmer and the software/machine configuration. The user's interface contains the mechanisms to produce queries and to handle such features as comment blocking, token completion, error handling facilities, etc. The programmer's interface provides the mechanisms to facilitate the development of new software for the lexical system (e.g. the I/O buffer; refer to the next section).

3.4.1 DEVICE INDEPENDENCE

The lexical system is designed as an input-device-independent machine. Input can be directed presently from three sources:

1. an I/O buffer,
2. a VMS file, and
3. the keyboard.

The I/O buffer is a stack. A stack was chosen rather than a queue because the stack functions already existed. When a string is to be loaded into the I/O buffer, a call is placed to a subroutine, passing as a parameter the string to be loaded. The string is then pushed onto the stack, character by character, right to left (backwards to simulate a queue). Input from the I/O buffer is given the highest priority; as

long as the stack is not empty, input is directed from this data structure. Hence, when modifying or adding modules to the Lexical System, programming situations requiring a particular character or string to be processed immediately are met by loading the character or string into the I/O buffer. Thus, as the Lexical System runs, regardless of what is happening elsewhere (e.g. user entering input via the keyboard), this character or string will be the next to be processed, followed by any input being entered or already entered via the other two input sources.

Previously defined queries may be stored in VMS files for later execution. A VMS file is given second highest priority. The user may specify that input is to be directed from a particular file and if the line buffer is empty, then input is taken from the file. This is analogous to "software pipes" and "include files".

The keyboard is given lowest priority. If the I/O buffer is empty and the user has not redirected the input to come from a VMS file, then input from the keyboard is processed. If the user continues to enter data from the keyboard while the input is being directed from the I/O buffer and/or a VMS file, the keyboard input is queued and processed only after the I/O buffer and/or VMS file is emptied. Regardless of the input device, all input is processed in the same manner.

Input device service priorities are assigned based on the criticality of the process requiring the various forms of input. The I/O buffer is given highest priority because particular routines critical to the

maintenance of lexical consistency and screen management have to utilize this data structure to function properly. For example, if a token is partially typed and will not fit on the remainder of the current line, the partial token is moved to the beginning of the following line. This is performed by popping a "screen" stack containing the sequence of characters on the current line and pushing these characters and a carriage return onto the I/O buffer. This is a pushdown automata consisting of the control (DFA table), the stacks, and the input stream provided by the I/O buffer, keyboard, and VMS file. Because the I/O buffer is no longer empty, the source of input will be automatically redirected from this data structure. The carriage return will be processed causing a new line to be initialized with the partial token following. This type of process must be executable at all times regardless of whether the input is currently directed from the keyboard or a VMS file. A VMS file is given second highest priority to prevent user input from the keyboard from disrupting the flow of input from the file.

The main significance of the I/O buffer is the impact created on the development of new software for the lexical system. If a programmer needs to disrupt the flow of input from the keyboard or a VMS file (note that the input becomes output to the output device) by generating additional output, the new software need only load the input into the I/O buffer (note again, that this input becomes the additional output to the output device). The consistency of the input processing mechanism is preserved because of device independence. The programmer does not

need to write additional software to process his/her "special" output. Note that "programming" refers to the actual development and implementation of software incorporated in the Lexical System; not a user-developed query.

3.4.2 SCREEN MANAGEMENT

The lexical analyzer is oriented around CRT-type terminal devices. The major philosophy in this work concerning screen appearance simply holds that only current information should remain visible. For example, only the current error message may appear on the screen. If the user response to the error message is itself in error, the old message is erased from the screen and the new error message is displayed. Likewise, after each prompt is correctly serviced, the screen is cleared before displaying the next prompt.

3.4.3 ERROR HANDLING

Lexical errors are detected on a character-by-character basis. For example, if a user intended to enter the command "SELECT", and instead entered "SEM", the first two characters would appear on the CRT screen since a command exists that begins with this two character sequence. However, if no token begins with the three character sequence "SEM", the character "M" will not be displayed on the CRT screen. Instead, an error handling facility will be called which incorporates three levels of error detection:

1. novice,
2. experienced, and
3. expert.

A novice user will always receive an error or warning message. An experienced user can optionally generate an error or warning message by entering the character "w" (why?) in response to the terminal bell (note that the character "w" generates on the output device "Why?? Because..."). At the expert level the terminal bell will not ring; nothing happens...that is, the character will not be displayed on the output device and the cursor remains in the current position. The three levels of error detection are not implemented; error or warning messages are always displayed.

When a message is displayed, the current command line is deleted from the screen and replaced with the error or warning message. The command line is then regenerated following the message. The important point is that the command line need never be retyped to eliminate lexical errors because these errors are the result of attempting to type a single character in a lexically invalid position (i.e. forms a string which is not the "prefix" of any token). The character never appears on the screen, so the user need not even backspace; the user need only type a correct character. The lexical system will continue as if nothing happened.

3.4.4 ADDITIONAL FEATURES

There are three additional major features incorporated into the lexical system:

1. No carriage return - the system automatically manages line characteristics ensuring that only full tokens fit within the line boundaries, if necessary, by moving tokens to the beginning of the following line, negating the end-user's need to use the return key. However, if a return key is depressed, appropriate processing will still take place.

2. Automatic comment blocking - comment delimiters will, at the user's request, automatically align themselves with the comment delimiters on the previous line, if the previous line contains comment delimiters and the comment is the last token on the line.
3. Facilities for "lazy" users - if the user does not wish to type the ending delimiters for comments, literals, identifiers, or numbers, the user need only depress the carriage return key and the system will automatically complete the processing of the token and begin a new line.

CHAPTER IV

4. TOKEN COMPLETION

A notable feature of the lexical system is token completion. As input is generated, a continuous effort is made to detect non-ambiguity in the current lexical environment. When this situation occurs, the token is automatically processed until the "new" current environment is again ambiguous (is unable to predict a prefix of the remainder of the token being analyzed). Note here that the two terms, "lexical environment" and "state of the lexical machine" (lexicon) are synonymous. Token completion is achieved with the use of an DDFA. For example, a user may intend to enter the token "ARCCOS". After accepting the first two characters "AR" from the user, the third character "C" may be automatically processed if and only if all tokens known in the lexicon, beginning with "AR" are followed by a "C" (non-ambiguity). Thus, entering "AR" will produce "ARC". Another "C" from the user may generate the remaining characters "OS" if and only if there is a single token in the lexicon beginning with "ARCC", namely "ARCCOS" (again, non-ambiguity). The user, then, need only enter "ARC" to produce "ARCCOS". If the user types the token "ARCCOS" in full, processing (including the echo) still produces the same token, namely "ARCCOS". Thus, token completion is available, but does not interfere.

The reverse process is termed token deletion. This process removes the current character from the screen along with any immediately preceeding characters that were automatically generated.

This paper does not promote the concept of token completion as a proven human factors concept. Rather, a tool is being provided to study this concept in detail in order to determine this concept's usefulness.

The following sections describe token completion beginning with a simple example. The concept will be augmented through the use of problems and solutions as a general working model is developed.

4.1 EXAMPLE

Suppose that a user decides to type the input "ARCCOS" and that the tokens "ARCCOS", "ARCTAN", "ABS", and "BTREE" are the only valid tokens in the lexicon. The user first types the character "A". The lexical analyzer determines that more than one valid token begins with this character, so nothing occurs other than echoing the character to the output device. The user next types the character "R". The lexical analyzer again determines that more than one valid token begins with the character string "AR" and nothing occurs other than echoing the new input character to the output device. The user next types the character "C". The lexical analyzer once again determines that more than one valid token begins with the character string "ARC", so again nothing occurs other than echoing the character to the output device. The user again types the character "C". At this point, the lexical analyzer determines that only one valid token begins with the character string "ARCC", namely, "ARCCOS". As a result, the character "C" is sent to the output device as well as the remainder of the token. As a result, the user need only type the characters "ARCC" in order to produce the token "ARCCOS". This process is termed token completion.

4.2 GRAPH-BASED DESCRIPTION

Figure 3 is a graph representing token completion. Each node in the graph represents a state of the lexical analyzer. The act of moving along an arc from one state to the next is referred to as a transition. The letters that label each arc are referred to as transition characters. A transition sequence is described by the following notation:

1a2r5,

which means that a transition occurred from state 1 to state 2 as a result of the transition character "A" and that a transition also occurred from state 2 to state 5 because of the transition character "R". Note that characters in this discussion will be capitalized for clarity and correspond directly with their lower case equivalents in the transition sequences. In the previous example with the token "ARCCOS" and from Figure 3, the following is a possible transition sequence:

1a2r5c6c7o8s9.

State 9, a state from which there are no possible transitions, is referred to as an end state. When an end state is reached, the current token has been completely recognized, processing has completed, and the lexical analyzer reinitializes to accept input for the next token.

Figure 4 illustrates the previous example of a user entering the character string "ARCCOS". The user first types the character "A", corresponding to the column in Figure 4 headed "INCOMING CHARACTER".

	POSSIBLE TOKENS	TRANSITION SEQUENCES
1)	abs	1a2b3s4
2)	arccos	1a2r5c6c7o8s9
3)	arcsin	1a2r5c6s10i11n12
4)	arctan	1a2r5c6t13a14n15
5)	btree	1b16t17r18e19e20

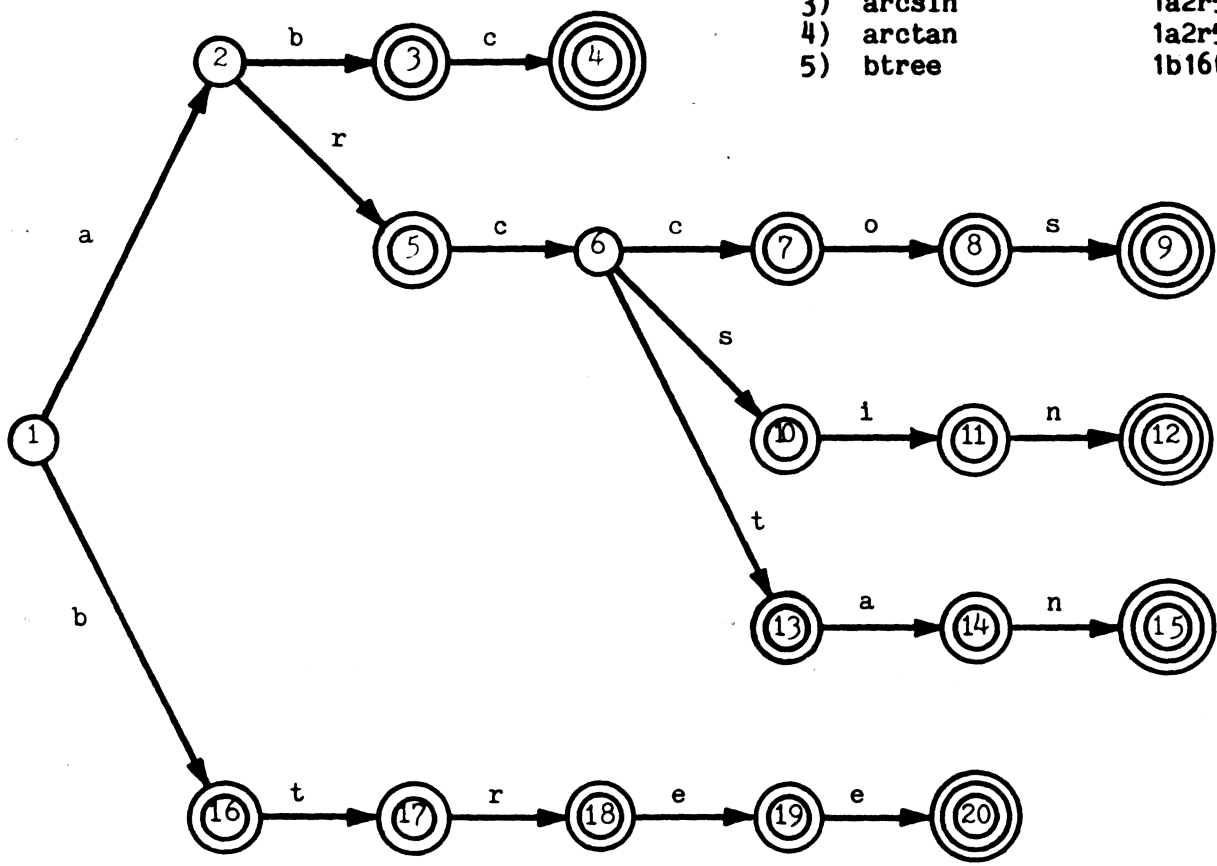


Figure 3: Graphical Representation of Token Completion

The lexical analyzer determines that this is a valid transition character, resulting in a transition from state 1 to state 2, corresponding to the word "PROCESS" in Figure 4 under the column headed "OPERATION". At state 2, the lexical analyzer determines that the only two possible valid characters the user may type is a "B" (for "ABS") or an "R" (for "ARCCOS", "ARCSIN", or "ARCTAN") since these two characters are the only ones that match transition characters permitting transitions from this state; otherwise an error results. If the user next types the character "R", the lexical analyzer matches this character with the transition character that causes a transition from state 2 to state 5. At state 5, the only valid character the user may type is a "C", and that character will be matched to the transition character that causes a transition from state 5 to state 6. If any other character is typed, an error results.

After the user types the second character "C", a transition occurs to state 7. From this state there is only one possible path that can lead to further state changes, namely state 7 to state 8 to end state 9. The transition characters that would ordinarily cause these transitions are "O" and "S". Since the lexical analyzer recognizes the uniqueness of this path, a state such as state 7 will be referred to as a recognizer state. The transition character causing a transition to a recognizer state is referred to as a recognizer character, and the unique path that leads out of a recognizer state is referred to as a recognizer path. Note that a recognizer character is a term applied to a data structure, in this case, a graph, to mark the start of a unique path. The character entered by the user, once matched into the graph as

PATH	INCOMING CHARACTER	REST OF TOKEN	OPERATION	COMMENT
1	a	rccos	process	
1a2	r	ccos	process	
1a2r5	c	cos	process	
1a2r5c6	c	os	process	recognizer
1a2r5c6c7o8s9			complete (os)	
1a2r5c6c7o8s9			recognize	"arccos"
1a2r5c6c7o8s9	o	s	throw out	endstate
1a2r5c6c7o8s9	s		throw out	endstate
1a2r5c6c7o8s9	<blank>		initialize	prepare for next token

OPERATIONS

complete: processes (see process operation) the incoming character and token complete the characters in parentheses, e.g. complete (os) means process the incoming character and complete the characters os.

process: uses the incoming character to determine the next state in the path.

recognize: determines that a valid token has occurred, namely the quoted token in the comment column.

throw out: discards the incoming character.

Figure 4: Token Completion

a valid transition or recognizer character, is a recognized character. The lexical analyzer knows when recognizer states have been reached and will echo the recognizer character to the output device and will also send to the output device the transition characters that define the transitions along the recognizer path. This is shown in Figure 4 with the word "COMPLETE" under the column headed "OPERATION". In short, when the user types the character string "ARCC", the fourth character "C" in the string causes a transition to a recognizer state, which is an indication that a unique token has been identified. The lexical analyzer takes control from the user, echoes the recognizer character to the output device, and traverses the recognizer path sending the transition characters to the output device. The end state marks the end of the recognizer path.

However, in this example the user has entered the entire character string "ARCCOS". The two final characters "OS" enter a pending state and are referred to as pending characters. The user action of typing pending and any additional characters is termed overtyping. Note that the potential for overtyping begins when a token has been recognized. To prevent overtyping, a delimiter character is required after a fully completed token. The lexical analyzer recognizes a blank as the delimiter. Once a token is completed, all immediately succeeding characters prior to the delimiter are thrown out. This is shown in Figure 4 by the words "THROW OUT" under the column headed "OPERATION". The delimiter is not part of the token which it delimits. In short, after a token completes, a delimiter (in this case, a blank) must be

typed in order to affirm the completion of the previous token and to reinitialize the lexical analyzer so that the next token may be processed (refer to "lexical endmarkers" in 10.4 Command Token Classes).

4.3 A FURTHER GENERALIZATION

Let us now generalize token completion one step further. If there is more than one transition leaving a state, these transitions will be termed alternate transitions. If only one transition is possible from a state, then that transition is termed a singular transition. Thus, in Figure 3, state 6 has three alternate transitions whereas state 5 has zero alternate transitions. Thus, a recognizer state is redefined as that state having only a singular transition and it follows that all states in a recognizer path are recognizer states. States with alternate transitions are referred to as decision states. Thus, in Figure 3 state 5 is a recognizer state and state 6 is a decision state. A recognizer state does not always cause a token to be completed entirely , but simply indicates that at this point in the input there is only one possible transition which will be automatically processed by the lexical analyzer. This transition may cause a state change to another recognizer state and if the rest of the token is represented by a recognizer path then the token is completed. This case is shown in Figure 5 in which state 5 is now a recognizer state.

Token completion does not necessarily imply a fully completed token. Partial token completion describes the case in which parts of a token are completed, as opposed to full token completion in which a token is fully completed. A set of rules have been defined that unify the two

concepts under one general concept, namely just token completion. This concept has the ability to generate one and only one character beyond the sequence of characters (prefix) that defines the current non-ambiguous environment (the current environment being defined as the set of possible tokens beginning with the prefix). The recursive nature of this definition is what allows the possibility of "apparently" token completing more than one character sequentially until the "new" current environment is once again ambiguous. The base case is a transition to a recognizer state. Each state in a recognizer path is a recognizer state. Thus, a recognizer path is a recognizer state plus a recognizer path. For this reason, characters that are sent to the output device as a result of token completion are referred to generally as token-completed characters.

A problem still remains because an incoming sequence of characters beginning with the three character string "ARC" will always be recognized by the lexical analyzer as the token "ARCCOS" regardless of the remainder of the sequence. Referring to Figure 6 the problem is that a transition has occurred from a recognizer state (recognizer character "R") to a decision state (token completed character "C") and back to a recognizer state (transition character "C") where the token completed character and the transition characters are equivalent (both "C"). The transition character that was entered by the user should have corresponded to the token completed character generated by the lexical analyzer. However, because the two characters are processed serially in priority order (token completed character first, user-entered character second), the character string "ARC" will cause the lexical analyzer to

PATH	INCOMING CHARACTER	REST OF TOKEN	OPERATION	COMMENT
1	a	rccos	process	alternate
1a2	r	ccos	complete (c)	primary recognizer state
1a2r5c6	c	cos	complete (os)	primary recognizer state
1a2r5c6c7o8s9			recognize	"arccos"
1a2r5c6c7o8s9	c	os	throw out	endstate
1a2r5c6c7o8s9	o	s	throw out	endstate
1a2r5c7o8s9	s		throw out	endstate
1a2r5c6c7o8s9	<blank>		initialize	prepare for next token

OPERATIONS

complete: processes (see process operation) the incoming character and token complete the characters in parentheses, e.g. complete (os) means process the incoming character and complete the characters os.

process: uses the incoming character to determine the next state in the path.

recognize: determines that a valid token has occurred, namely the quoted token in the comment column.

throw out: discards the incoming character.

Figure 5: Token Completion

process the character string "ARCC". This string is sufficient to complete to the token "ARCCOS" with all immediately succeeding non-blank characters being thrown out. While this will allow the token "ARCCOS" to be completed correctly, any character string that begins with the prefix "ARC" will always produce the token "ARCCOS".

A possible solution is to design the lexical analyzer to recognize the above situation and to throw out the user-generated transition character if and only if this character and the token completed character are equivalent. As will be shown, this is only a partial solution, but at least the character strings "ARCCOS", "ARCSIN", and "ARCTAN" can again be typed and have the tokens "ARCCOS", "ARCSIN", and "ARCTAN" respectively sent to the output device. This solution is shown in Figure 7 with the word "MATCHED" under the column headed "COMMENT". At this point, the incoming character "C" is thrown out since it matches the previously token completed character. In this manner, when the user types "ARCSIN", the token "ARCSIN" will be recognized. However, if the user now tries typing the character string "ARC", that is all that is sent to the output device, namely "ARC". This occurs because the transition character "C" generated by the user is matched and thrown out as shown in Figure 7, rather than causing a token completion to "ARCCOS" as in Figures 5 and 6. In short, the lexical analyzer is left in a decision state. However, the full solution follows.

In figure 8 at the point where the character "C" is matched and thrown out, the user has thus far typed the character string "ARC" and the character string "ARC" also appears on the output device. In order for the character string "ARC" to token-complete to the token "ARCCOS", a

PATH	INCOMING CHARACTER	REST OF TOKEN	OPERATION	COMMENT
1	a	rcsin	process	alternate decision state
1a2	r	csin	complete (c)	primary recognizer state
1a2r5c6	c	sin	complete (os)	primary recognizer state
1a2r5c6c7o8s9			recognize	"arccos"
1a2r5c6c7o8s9	s	in	throw out	endstate
1a2r5c6c7o8s9	i	n	throw out	endstate
1a2r5c6c7o8s9	n		throw out	endstate
1a2r5c6c7o8s9	<blank>		initialize	prepare for next token

OPERATIONS

complete: processes (see process operation) the incoming character and token complete the characters in parentheses, e.g. complete (os) means process the incoming character and complete the characters os.

process: uses the incoming character to determine the next state in the path.

recognize: determines that a valid token has occurred, namely the quoted token in the comment column.

throw out: discards the incoming character.

Figure 6: Token Completion

delimiter has to be typed. The reasoning is as follows: the character strings "ARS" and "ART" token-complete to the tokens "ARCSIN" and "ARCTAN", respectively (refer to Figure 7 substituting "ARCTAN" for "ARCSIN"). In addition, the character strings "ARCSIN" and "ARCTAN" also token-complete to the tokens "ARCSIN" and "ARCTAN", respectively (again, refer to Figure 8), but only because the character "C" typed by the user is thrown out. And again, the character string "ARCCOS" token completes to the token "ARCCOS" even though the character "C" is thrown out, because the next character is also a "C" (refer to Figure 8 substituting "ARCCOS" for "ARCSIN"). This is not the case with the character string "ARC". Once the character "C" is thrown out, the lexical analyzer remains in a decision state until further input is received. However, the only character that can cause a transition after a token has completed is a delimiter. If a delimiter is indeed the next character to be processed after the character string "ARC" has been processed, the implication is that a character "C" was thrown out, further implying that the user did indeed type the character string "ARC" intending for the token "ARCCOS" to be produced. Any character other than a "C" would not be thrown out following token completion (remember "ART" and "ARS") and processing would continue. Therefore, a delimiter signifies that the end of the character string has been reached, namely "ARC", and will permit the additional transitions to the end state. As can be seen, a problem does not really exist, but rather, in this particular situation, the lexical analyzer cannot identify the token until after the delimiter is typed. In all other cases, the token is identified prior to typing the delimiter.

PATH	INCOMING CHARACTER	REST OF TOKEN	OPERATION	COMMENT
1	a	rcsin	process	alternate decision state
1a2	r	csin	complete (c)	primary recognizer state
1a2r5c6	c	sin	throw out	matched
1a2r5c6	s	in	complete (in)	primary recognizer state
1a2r5c6c7o8s9	i	n	recognize	"arcsin"
1a2r5c6c7o8s9			throw out	endstate
1a2r5c6c7o8s9	n		throw out	endstate
1a2r5c6c7o8s9	<blank>		initialize	prepare for next token

OPERATIONS

complete: processes (see process operation) the incoming character and token complete the characters in parentheses, e.g. complete (os) means process the incoming character and complete the characters os.

process: uses the incoming character to determine the next state in the path.

recognize: determines that a valid token has occurred, namely the quoted token in the comment column.

throw out: discards the incoming character.

Figure 7: Token Completion

PATH	INCOMING CHARACTER	REST OF TOKEN	OPERATION	COMMENT
1	a	rc	process	alternate decision state
1a2	r	c	complete (c)	primary recognizer state
1a2r5c6	c		throw out (c)	matched
1a2r5c6	<blank>		complete (os)	primary recognizer state
1a2r5c6c7o8s9			recognize	"arccos"
1a2r5c6c7o8s9			initialize	prepare for next token

OPERATIONS

complete: processes (see process operation) the incoming character and token complete the characters in parentheses, e.g. complete (os) means process the incoming character and complete the characters os.

process: uses the incoming character to determine the next state in the path.

recognize: determines that a valid token has occurred, namely the quoted token in the comment column.

throw out: discards the incoming character.

Figure 8: Token Completion

CHAPTER V

5. SOFTWARE STANDARDS IN THE LEXICAL SYSTEM

The implementation of the lexical process is divided into two categories:

1) system design (intermodular design), and
2) module design (intramodular design).

Each category has associated standards whose purposes are to maintain a system that is consistent in various aspects.

5.1 INTERMODULAR DESIGN

In the area of intermodular design, the standards deal with the functional qualities of modules. These standards are as follows:

1) each module serves one and only one function,
2) each module, when requiring the services of another function, calls the module associated with that function, and
3) a module is defined for a function when the services of that function are required at more than one point in the system.

These concepts are by no means new and are generally referred to as modularization. For example, the operation of concatenation is a string function whose services are required at many points in the system. Therefore, a module has been defined whose only purpose is to perform this operation. The concatenation module in turn requires the services of another operation which determines the length of a given character string. The services of this "length" operation are also required at various other points within the system. Hence, the length operation is

a function for which a "length" module has been defined. This definition of a "one module/one function" concept leads to an extremely high degree of modular code in which all but the most basic functions consist solely of three principle language constructs:

1) IF...THEN...ELSE...ENDIF,
2) DO <WHILE>...ENDDO, and
3) calls to other functions.

The basic functions, such as string manipulations and I/O operations, are grouped together as a class of functions referred to as "system primitives" and are generally more machine dependent.

5.2 INTRAMODULAR DESIGN

Intramodular design deals with the coding style of each module to ensure that their format is the same irrespective of the author. The coding standards are as follows:

1) all keywords (IF, THEN, ELSE, ENDIF, DO, WHILE, etc.) are capitalized,
2) each variable name is to be fully mnemonic with no "personalized" abbreviations permitted. Taking full advantage of the compiler's limit of thirty-two characters is highly recommended,
3) indentation is three spaces beginning in column seven,
4) IF...THEN...ELSE...ENDIF takes the form:

```
IF <condition>
   THEN
<blank line>
      [block]
<blank line>
   ELSE
<blank line>
      [block]
<blank line>
ENDIF,
```

5) DO <WHILE>...ENDDO takes the form:

```
DO <WHILE> <condition>
```

```
<blank line>
   [block]
<blank line>
END DO,
```

6) the declaration portion of a module utilizes the following format in the order given:

   a) character declarations in ascending order on the number of bytes,
   b) integer declarations in ascending order on the number of bytes,
   c) logical declarations,
   d) INCLUDE files,
   e) DATA statements,

7) label numbers for I/O format statements are two digits starting at ten and ascending by ten. Label numbers for END_OF_FILE and OPEN error conditions are three digits starting at 100 and ascending by 100. All labels are right justified to column five,
8) a blank line precedes each END statement,
9) lists of variables in declaration and/or data statements are in alphabetical order,
10) INCLUDE statements are in alphabetical order by INCLUDE file name where appropriate,
11) when continuing conditions of conditional statements (i.e. IF) on additional lines, the left most parenthesis will be aligned with the left most parenthesis of the previous line and the text of the condition will be left justified and aligned with the text of the previous line. This may require the insertion of blank characters between the right most left parenthesis and the start of the text, and
12) modules are no more than two pages in length (most are no more than one page).

5.3 CLASS CONCEPT

The modules of the lexical process are divided into several classes which are as follows:

1) pseudo-parsing subsystem,
2) lexical-driver subsystem,
3) pre-lexical subsystem,
4) I/O subsystem,
5) delete/parser-backup subsystem,
6) token completion subsystem,

   a) token-completion-on, and
   b) token-completion-off,

7) post-lexical subsystem,
8) string subsystem, and
9) stack subsystem.

Each module in a class performs a single function that is related to the concept of that class. For example, in the I/O subsystem, there are five major modules. Each module performs one of the following functions:

1) determine the source of input,
2) process an END_OF_LINE condition,
3) acquire input from the terminal,
4) acquire input from the buffer, and
5) acquire input from a file.

These are five distinct functions that are performed by five disjoint modules. Yet, each function is I/O related. These functions, in turn, request services from other functions, some of which may not be of the same class. For example, all five major I/O related functions require services from the string and stack subsystems. Below is a brief description of each class.

5.3.1 PSEUDO-PARSING SUBSYSTEM

The pseudo-parsing subsystem is the top level subsystem consisting of a group of modules that serves the basic function a parser would ordinarily serve. This subsystem is temporary and will be eliminated when the LALR(1) parser becomes operational. The pseudo-parsing subsystem initializes the user's terminal environment either through a predefined profile or through a series of prompts to the user. Such parameters as terminal screen size and comment blocking are thereby

initialized. Once the terminal environment is defined, a call is placed to the initializing routine that opens the command, noiseword, and synonym token files and builds the lexical forest of generalized trees utilized by the DDFA. After initialization has been completed, the pseudo-parser subsystem enters a loop in which a call is placed to the lexical-driver subsystem from which a return is made at the completion of each token. After the return, the call to the lexical-driver subsystem is repeated, ad infinitum, until the user enters the token "quit", at which time the pseudo-parser subsystem terminates execution.

5.3.2 LEXICAL-DRIVER SUBSYSTEM

The lexical-driver subsystem is called by the pseudo-parser subsystem. The lexical-driver subsystem calls every other subsystem, except the psuedo-parser. The lexical driver is the driving mechanism for lexical analysis. Calls are invoked to initialize the lexical process, acquire input, analyze the input, generate error and/or warning messages, handle error recoveries, token-complete, "house clean" after recognizing a complete token, and return the necessary information back to the pseudo-parser subsystem.

5.3.3 PRE-LEXICAL SUBSYSTEM

The pre-lexical subsystem is called by the lexical-driver subsystem. At this point, all data structures not initialized by the pseudo-parser subsystem are initialized in preparation for accepting a new token. The data structures are primarily stacks and their pointers, utilized to "remember" the lexical information about the

current terminal line, especially when pertaining to the deletion of what has been previously entered.

5.3.4 I/O SUBSYSTEM

The I/O subsystem is called by the lexical-driver subsystem. The I/O subsystem returns a single character and the ASCII hexadecimal value for that character. The source of input is determined and terminal I/O and some error conditions are processed. This subsystem is responsible for repositioning a partial token to the beginning of the next line when the END_OF_LINE condition is encountered.

5.3.5 DELETE/BACKUP-PARSER SUBSYSTEM

The delete/backup-parser subsystem is called by the lexical-driver subsystem and is responsible for processing the characters being deleted and for placing a call to a module that is responsible for "backing up" the parser to a prior state when a token has been completely deleted. Deletion requires numerous stack operations, terminal I/O processing, and pointer reinitializations to return the system to the state that is consistent with the information remaining on the current terminal line after the delete operation. The body of this module is not implemented. That is, the module exists but is only the target of a "dummy" call.

5.3.6 TOKEN-COMPLETION SUBSYSTEM

The token-completion subsystem is called by the lexical-driver subsystem and is divided into two sections:

1) token-completion-off, and

2) token-completion-on.

The token-completion-off section utilizes a DFA table and is executed when defining identifiers, literals, numbers, or comments. Once the definition is completed, the identifier, literal, or number is inserted into the lexical forest for future token completion. Comments are of course not token-completed.

The token-completion-on section is executed when dealing with tokens that are already defined in the lexical forest. The transition between these two sections is controlled by the post-lexical subsystem (see section 5.3.7).

Both sections determine whether or not a token has been completed. If completion has not occurred, then processing returns to the lexical-driver subsystem into a loop and the calls to the other subsystems are repeated. If a token is completed, a return is made to the lexical-driver subsystem, the loop is terminated, the post-lexical subsystem is called and finally a return is made to the pseudo-parsing subsystem.

5.3.7 POST-LEXICAL SUBSYSTEM

The post-lexical subsystem is called by the lexical-driver subsystem and performs the "house cleaning" functions after a token has been identified. This subsystem is responsible for determining the token value and the address of the descriptor for variables, as well as performing automatic comment blocking and the back and forth switching between the two sections of the token-completion subsystem.

The switching mechanism is controlled within this subsystem by inspecting the value of a variable that is set to "true" if token completion is off and "false" otherwise (refer to 6.2.4 DFA_CODE STACK).

5.3.8 STRING AND STACK SUBSYSTEMS

The remaining two subsystems, string and stack, consist of support functions to perform PL/I-like string manipulations and stack operations. Requests for services from these two subsystems are made from virtually every module in the system and are the most heavily utilized.

CHAPTER VI

6. DATA STRUCTURES

The lexical analysis operations of the lexical system utilize two primary data structures:

1) a forest of generalized trees, and
2) a set of stacks.

The forest of generalized trees is referred to as the lexical forest and is the basis for token completion. The set of stacks is referred to as the reverse lexical stacks and is the basis for token deletion.

6.1 THE LEXICAL FOREST

The lexical forest is a forest of generalized trees implemented as binary trees. Each parent contains two sibling pointers. One pointer links to the next lower level and the other pointer links to nodes on the same level as the parent (Figure 9). If Figure 9 is rotated 45 degrees to the right, the node labeled 1 becomes the root of an unbalanced binary tree. The lexical forest is the implementation of the graph as described in the graph-based discussion of token completion.

6.1.1 NODES

Each node of the lexical forest contains six items of information:

1) the transition character,
2) a link to the next level,
3) a link to the next node at the same level,
4) the concatenated recognizer path,
5) the keyword value, and
6) the token value, or, descriptor address.

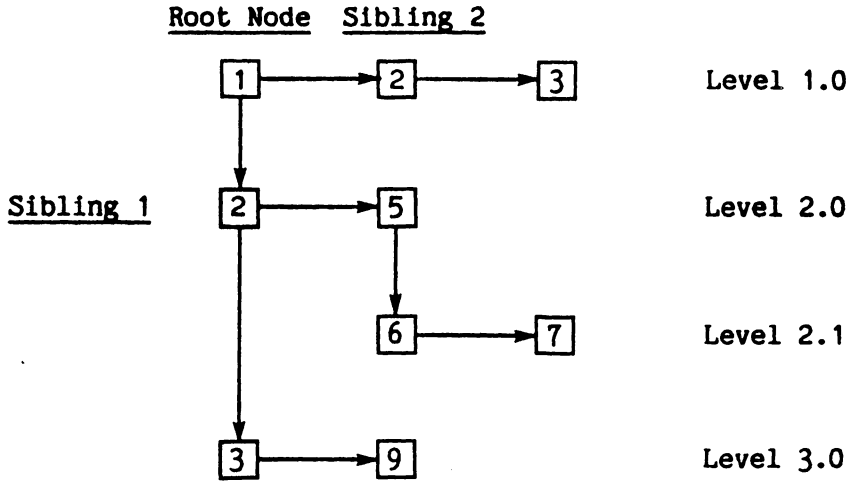


FIGURE 9: The Lexical Forest represented as a forest of general trees.

In order to determine if the transition to the next state is valid or not, the following steps are taken:

1) accept the input character from the terminal device,
2) compare the input character to the transition character stored at the node linked at the next level (referred to as the next state),
3) if the comparison in 2) is false, compare the input character to the transition character stored at each successive node at the same level (referred to as the alternate state),
4) if a match is found in step (2 or (3 then the transition is valid; else an error has occurred.

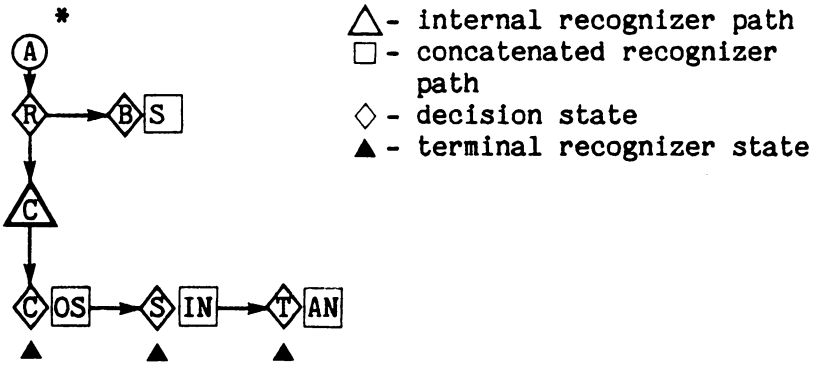
Note that a node is a decision state if that node is linked to a node at the next immediate level which is in turn linked to a node at the same level; all other nodes are recognizer states. Nodes with no links to the next level are terminal end states. Thus, recognizer paths descend downward to lower levels (tree skewed left); decision states remain at the same level (non-empty right subtree).

Some recognizer paths do not entirely complete a token. Such paths are called internal recognizer paths. When a recognizer path terminates at a terminal end state, the recognizer character of each transition along the recognizer path is actually stored as a concatenated string at the node corresponding to the recognizer state. This contrasts with internal recognizer paths in which the recognizer characters are stored at each node along the recognizer path. Thus, the implementation using general trees is a restricted form of the graph-based discussion in which all recognizer characters are stored at each node of the recognizer path regardless of whether or not the path is internal. (Figure 10). This restriction is a performance issue and is intended to

reduce the number of links in recognizer paths in order to increase the speed of token completion (refer to section 11 Performance Issues And Miscellaneous Problems). In short, token completion occurs when no link exists to a node at the same level or a terminal end state is encountered. Token completion that does not entirely complete a token is referred to as internal token completion. The recognizer state is referred to as an internal recognizer state and the recognizer path is referred to as the internal recognizer path. The characters echoed to the screen are referred to as internal token completed characters. Token completion that completes a token is referred to as terminal token completion. The terminal end state is referred to as the terminal recognizer state and the recognizer path is referred to as the concatenated recognizer path. The characters echoed to the output device are referred to as terminal token completed characters (Figure 10). Notice that internal token completion is the implementation of partial token completion and terminal token completion is the implementation of full token completion. In the graph-based discussion, there is no distinction; that is, reference is made only to token completion (refer to 4.2 A FURTHER GENERALIZATION). However, in the implementation, the two types of completions are implemented differently so must both be referenced.

6.1.2 POINTER ARRAYS

Entry into the lexical forest is controlled by an array of pointers. There is a general tree for each class of tokens beginning with the same character. Thus, there is a tree for all tokens beginning with the letter "A". The array of pointers contains a pointer for each class of



*
Note that the root node always has one sibling since the entry point is defined and by definition cannot be a decision state. The collection of root nodes of the general trees of the lexical forest are actually alternate states at the same level of an imaginary tree root.

Figure 10: Graphic-based implementation of the Lexical Forest

tokens. The pointer points to the root node of the general tree for that class of tokens.

The proper index into the pointer array is controlled by the array of pointer characters in which the starting character for each class of tokens is stored. The sequential position of the starting character in the array (the index) is the proper index into the pointer array (Figure 11). These two pointer arrays can be thought of as the lexical forest root node.

6.1.3 THE CHARACTER SET AND FOREST INSERTION

The character set comprises all ASCII characters and is stored in an array. A token is added into the lexical forest character by character. The first character is indexed into the character set array. If a match is made, then the character is a legal character of the lexical system. The first character is next indexed into the array of pointer characters to determine the index of the pointer array from which the address (index) of the root node may be determined of the general tree for the class of tokens beginning with that character. Each additional character of the token is only indexed into the character set array. If no general tree currently exists for this class of tokens (i.e., this is the first token of this class), the first character must be added to the array of pointer characters, a root node allocated, and the address of the root node added to the pointer array. Once the address of a root node has been determined, the token may be inserted into the general tree.

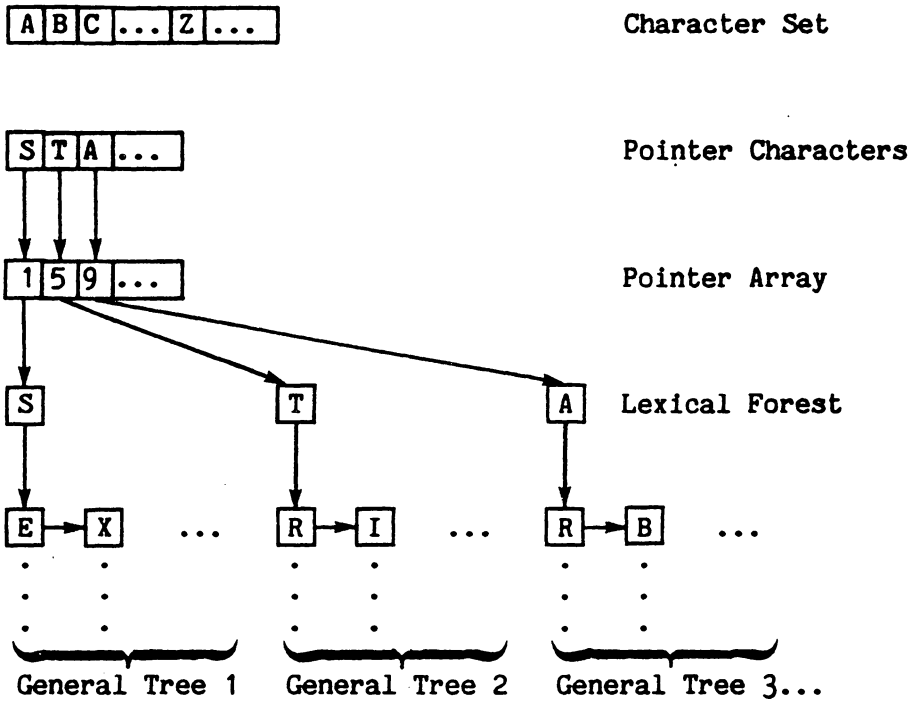


Figure 11: Entry points into the Lexical Forest
(Lexical Forest Root)

6.1.4 TOKEN COMPLETION

Token Completion is implemented with three pointers:

1) a current node pointer,
2) a total knowledge pointer, and
3) an ignorant pointer.

All three pointers point to nodes within the lexical forest. More precisely, the pointers point to nodes in the same general tree for the class of tokens corresponding to the token the user is entering. The current node points to the node that corresponds to the last character echoed to the output device. The total knowledge pointer points to the node that corresponds to the last transition due to token completion; in other words, this pointer is synonymous to a user that has total knowledge of the command completion process and does not type any unnecessary characters in defining the token. The ignorant pointer points to the node corresponding to the last transition assuming that token completion does not exist; in other words, this pointer is synonymous to a user that is ignorant of the token completion process and does not rely on this process for any characters.

As a token is entered, each character is processed causing three independent transitions. The transitions and their pointers form distinct, overlapping, and possibly, but not necessarily, identical transition sequences. When the token has been completed and recognized, the three transition sequences and the three pointers will be equivalent. The three pointers are utilized to implement the solutions to the problems examined in the graph-based discussion.

As the three transition sequences grow independently and at different rates, certain interactions take place to ensure that the final transition sequences and pointers are equivalent. These interactions obey the following rules for each input character entered (each rule is further developed in the paragraphs that follow):

1) The general tree is traversed with token completion until encountering a decision state or a terminal end state. The total knowledge pointer is assigned this state number,
2) If the ignorant pointer and the current node are equal, the general tree is again traversed with token completion until a decision state or terminal end state is encountered. The current node is assigned this state number and the transition character is echoed to the output device. The ignorant pointer is assigned the state number of the node after the first transition, and
3) If the ignorant pointer and the current node are not equal, then the ignorant pointer is re-evaluated for each input character until an input character is encountered that is not equal to any previously internal token completed character or the two pointers are again equal.

If a transition does not exist when attempting to process an input character and a terminal recognizer state has not been encountered, the input character is invalid and an error condition exists. Also, a state number is simply the index of the node in the lexical forest.

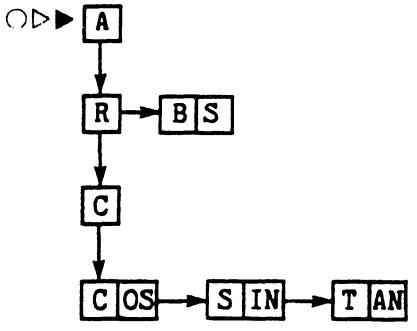
The first rule permits the processing of an internal end state. In addition, if an input character is found to be illegal, yet, this input character is a delimiter and the total knowledge pointer points to a terminal end state, then the token is completed. Terminal end states have a value of zero assigned to the pointer that links them to the next level.

The second rule is evaluated as a result of internal token completion having occurred due to the previous input character. The inequality between the ignorant pointer and the current node is a signal that there exists a potential for ambiguity between the user-induced and token completion-induced transitions (Figure 12).

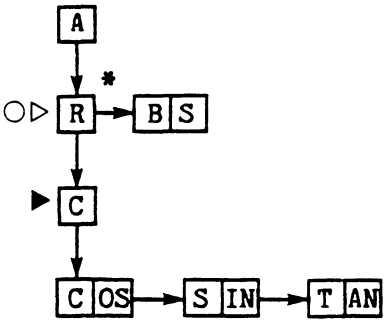
The third rule resolves the ambiguity detected by the second rule. As the ignorant pointer is advanced through the general tree for each input character, the current input character is compared to the top value of a stack containing the previously internal token completed characters that generated the inequality. If a match is made, the input character is discarded and the ignorant pointer is evaluated for that single transition. This process continues until a stack entry is encountered that does not match the input character or the stack is emptied. Note that the stack depth is the number of states separating the ignorant pointer and the current node. Therefore, when the stack is empty, the two pointers are once again equal. If an inequality is encountered between a stack entry and the input character, the assumption is that the user is taking advantage of token completion. Therefore, the ignorant pointer is updated to the value of the current node (Figure 13). Regardless of whether an inequality is detected or the stack is emptied, the ignorant pointer and the current node are once again equal. The point at which input is again processed using rule 1 above is being controlled by rule 3.

Step 1: User Typed "A"

▷- ignorant pointer
○- current pointer
▶- total knowledge pointer



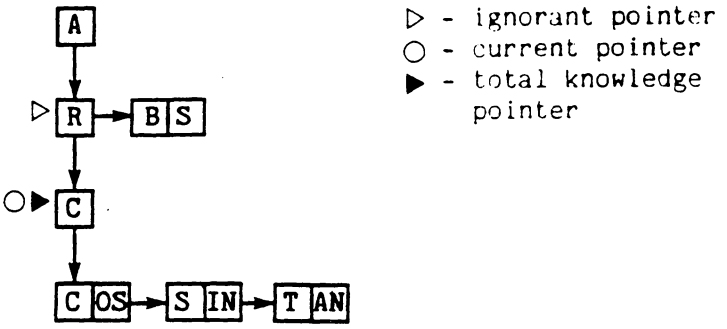
Step 2: User Typed "R"



*
Internal token completion is accomplished by loading into the I/O buffer the transition character at the next level down after the current internal recognizer state has been processed.

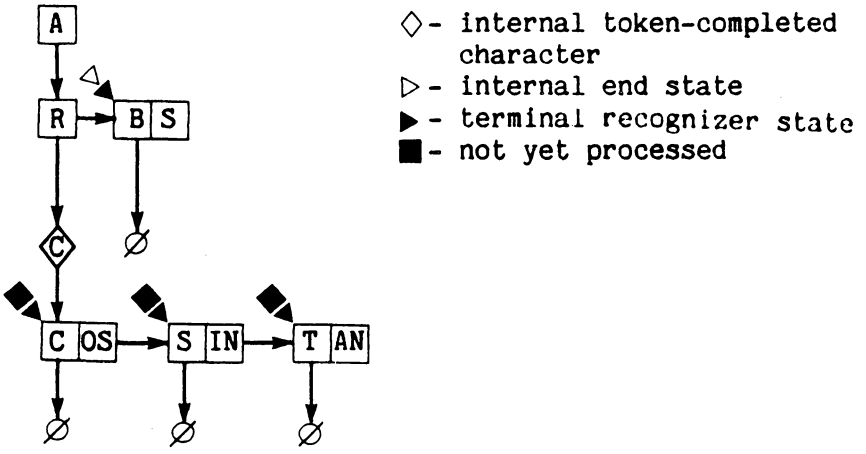
FIGURE 12: Token Completion Transitions
(continued next page)

Step 3: Internal Token Completion



Note that pointers are advanced for each keystroke. There have been two key strokes. If the user was correctly typing "ARCCOS" and token completion did not exist then "AR" would be currently on the screen (ignorant pointer). If the user was taking advantage of token completion, two keystrokes ("AC") would yield "ARCCOS" (total knowledge pointer).

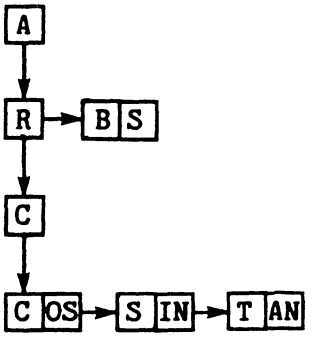
FIGURE 12: Token Completion Transitions
(continued next page)



Potential for ambiguity; if the next input character is a "C" , is this the first "C" (novice user) or the second "C" (experienced user)?

Figure 12: Token Completion Transitions

General Tree



Partial Completion Stack

Example 1: User type "C" = top of partial completion stack; stack popped; evaluate ignorant pointer; input character discarded; stack empty (ignorant pointer = current node); process next input character.

Example 2: User types "S" ~= top of partial completion stack; input character loaded into I/O buffer; partial completion stack cleared; set ignorant pointer = current node. Note that the input character is an alternate path requiring the rerouting of the total knowledge pointer. In practice, this is not done for terminal recognizer states due to performance degradation.

Figure 13: Resolving Ambiguity

6.1.5 TOKEN COMPLETION SUMMARIZED

There are three transition sequences. The first corresponds to the line of output (current node and output sequence). The second corresponds to an inexperienced user (ignorant pointer and novice sequence). The third corresponds to an expert user (total knowledge pointer and expert sequence).

The novice sequence can lag behind and at most equal the output sequence. The expert sequence is at least equal to but usually greater than the output sequence. The expert and output sequences become greater than the novice sequence and the expert sequence becomes greater than the output sequence because internal and/or terminal token completion has occurred.

When the novice sequence lags behind the output sequence, the token completion process must stop to permit the novice sequence to "catch up" with the output sequence. This is due to the assumption that the inexperienced user knows nothing about token completion and will continue to enter those characters that were just token completed.

The expert sequence always forms a transition sequence with token completion using the next state links. However, if the current sequence uses an alternate state link, then the expert sequence will have to be "backed up" and rerouted along the alternate path. This situation never arises with the novice sequence since this sequence can never be greater than the output sequence. The expert sequence is always the first to arrive at a terminal end state which permits the recognition of tokens when a delimiter is entered. Remember that this is the only

circumstance in which a token is not recognized until after the delimiter has been entered.

6.1.6 KEYWORD VALUES AND DESCRIPTORS

Keyword values, e.g. command names, were discussed earlier in the paper. Descriptors pertain to identifiers, e.g. attribute or relation names. Identifiers are stored in the lexical forest after they have been defined, to permit their subsequent token completion. Rather than being assigned a keyword value, identifiers are assigned a pointer that serves as an index into a data structure (part of the data definition for the DBMS) that contains the following information:

1) type (boolean, character, integer, or real),
2) length,
3) value, and
4) possibly some indexing information.

In this manner, as soon as an identifier is recognized, the pointer to the descriptor is also available since this pointer is stored in the keyword value field of the lexical forest.

6.2 REVERSE LEXICAL STACKS (PUSHDOWN AUTOMATON)

The reverse lexical stacks are a set of five stacks used to "remember" what has been sent to the output device, using the concept of a pushdown automaton. Each stack can retain one line of output up to 132 characters long. A successful transition from one state to the next in the lexical forest results in information about the current node being stacked onto the reverse lexical stacks.

6.2.1 STACK FUNCTIONS

The purpose for each stack is as follows (unfortunately, the concept of "token completion" was developed after writing several thousand lines of FORTRAN in which the variable names refer to "command completion". Since this section of the paper refers to the implementation, all references to token completion reflect the actual variable names, i.e. command completion):

1) LEXPATH - to stack the state number (node address in the lexical forest) of each character echoed to the output device,
2) LINE - to stack each character echoed to the output device,
3) DFA_CODE_STACK - to stack the DFA code for each character of a token,
4) CMD_COMPLETION_OFF - to stack whether terminal token completion was operative or inoperative for each character of a token,
5) PARTIAL_CMD_COMPLETION_STACK - to stack whether internal token completion was operative or inoperative for each character of a token.

6.2.2 LEXPATH

Lexpath is used to resume processing input after a deletion has occurred. If a user has entered a token and then deletes back to the middle of the token, the lexical analysis operation needs the new starting address (state) within the lexical forest or the DFA table from which input processing may resume. Therefore, each deletion results in a popping of LEXPATH to expose the new starting address. Blanks between tokens are assigned a state value of zero. After a deletion is completed and if a zero is on the top of LEXPATH, then continued processing of input begins at the root of the lexical forest. Lexpath is initialized with a zero when the stack is emptied and when a token is completed.

6.2.3 LINE

LINE is used to speed the operation of regenerating a token or an entire line of tokens. When an uncompleted token encounters the END_OF_LINE condition, the characters entered so far must be popped from the reverse lexical stacks and pushed onto temporary reverse lexical stacks. The corresponding characters are deleted from the output device. A new line is initialized, the reverse lexical stacks are cleared and initialized, and finally, the temporary stacks are pushed back onto the reverse lexical stacks. As each entry from the temporary stacks is popped and before being pushed back onto the reverse lexical stacks, the value for LINE is first echoed to the output device, negating the need to access and traverse the lexical forest. This ensures the disjointness of the operations of token completion and token deletion in which each operation has its own set of data structures typed specifically for that one operation. Error processing may require partial or complete line regeneration. LINE is initialized with a null character when the stack is cleared or when a token is completed.

6.2.4 DFA CODE STACK

The DFA_CODE_STACK retains the DFA code information. The DFA code takes on one of five possible values:

1. one (1) = literal,
2. two (2) = identifier,
3. three (3) = comment,
4. four (4) = a number beginning with a digit, and
5. five (5) = a number beginning with a decimal.

A DFA code is assigned to every token. However, the DFA code has meaning only for the four command tokens utilized to define new tokens. These are:

1. for literal there is the beginning delimiter "'",
2. for identifier there is "BOOLEAN", "CHARACTER", "INTEGER", "REAL", and
3. for comment there is the beginning delimiter "/*".

The definition command tokens, "INTEGER" and "REAL", include numbers that begin with digits or decimals. The DFA code is reset only if one of these definition command tokens is encountered. The value does not change after the definition is complete nor for any additional tokens unless that token is also a definition command token. During definition, token completion is turned off and further input processing involves a DFA table rather than the lexical forest. Entries onto LEXPATH are no longer states from the lexical forest but are instead states from the DFA table. When the definition is complete, token completion is turned back on and input processing once again involves the lexical forest (Figure 14).

6.2.5 COMMAND COMPLETION OFF STACK

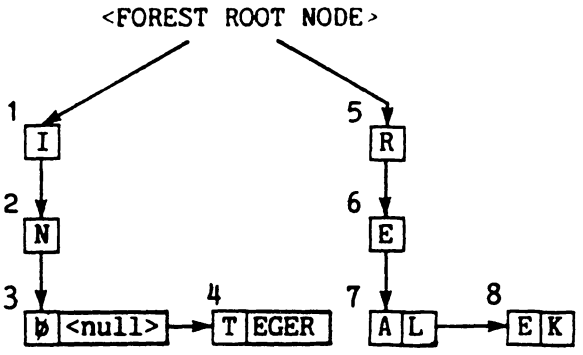
The command completion off stack is used to store whether or not token completion is operational. When input processing interacts with the lexical forest, a "false" value is pushed onto the stack; otherwise a "true" value is pushed onto the stack, indicating that the DFA table is being utilized.

6.2.6 PARTIAL CMD COMPLETION STACK

Partial command completion is set to "true" only when the corresponding character in LINE was internally token completed. Thus, in Figure 14, a "true" value is pushed onto PARTIAL_CMD_COMPLETION_STACK when the character "E" in the token "REAL" is encountered. Notice the distinction between terminal token completion and internal token completion. Terminal token completion defaults to "on" and is only turned "off" if the dfa code is reset due to a definition command token, indicating that there will be no interactions with the lexical forest. This contrasts to internal token completion which defaults to "off" and is only turned "on" if a recognizer state is encountered.

6.2.7 TOKEN DELETION

Token deletion requires a process which is somewhat the inverse of token completion. When a deletion is processed, a character is removed from the output device along with all preceeding token-completed characters. The reverse lexical stacks must be popped appropriately. The length of the top entry in LINE is used to determine how many characters are to be erased from the output device. This permits the removal of the echoed concatenated recognizer path of a token from the output device. Further, as long as the top of the PARTIAL_CMD_COMPLETION_STACK is "true", then deletion continues until a "false" value is once again encountered. This permits the deletion of internally token-completed characters from the output device. If the top of "COMMAND_COMPLETION_OFF" is "true" (meaning that token completion is off) then the new current lexical environment, after the deletion



User types: REAL number (number is a variable name)

lexpath	line	dfa_code_ stack	cmd_completion_ stack	partial_cmd_ completion stack
0	<null>	2	false	false
12	<blank>	2	true	false
4	R	2	true	false
4	E	2	true	false
4	B	2	true	false
4	M	2	true	false
4	U	2	true	false
4	N	2	true	false
0	0	2	true	false
7	L	0	false	false
7	A	0	false	false
6	E	0	false	true
5	R	0	false	false
0	<null>	0	false	false

Figure 14: Reverse Lexical Stacks

process has completed, is to resume using the DFA table and not the lexical forest. The top of LEXPATH is the state at which processing is to begin and the top of DFA_CODE_STACK is the type of token being entered (literal, identifier, comment, number beginning with a digit, or number beginning with a decimal). In short, these five stacks contain all of the necessary information to resume processing from a new lexical environment after completing one or more deletion operations.

CHAPTER VII

7. STACK MANAGEMENT

Stack management is array-based. There are two 25 X 300 matrices declared, each of which allows for up to 25 stacks of 300 entries per stack. There is a matrix for integer stacks (e.g., LEXPATH) and a matrix for character stacks (e.g., LINE). Associated with each matrix is a one-dimensional pointer array whose entries are pointers to the tops of each stack. Each stack is assigned a mnemonic variable name (e.g., LEXPATH and LINE) which is in turn assigned an integer value ranging from one to twenty-five. The integer values serve as indices into the pointer arrays where the index of the top of the corresponding stack may be located in the matrices (Figure 15).

Boolean stacks are treated as integer stacks, with an integer "1" value representing "true" and an integer "0" representing "false". There are several additional stacks. In particular, there are stacks that will eventually be utilized by the LALR parser. For a description of each stack, refer to Appendix B.

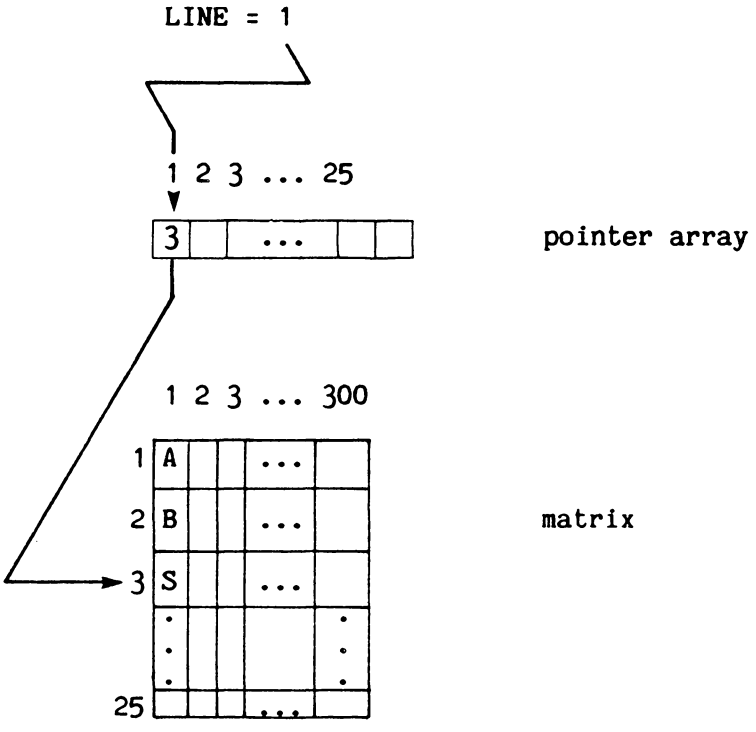


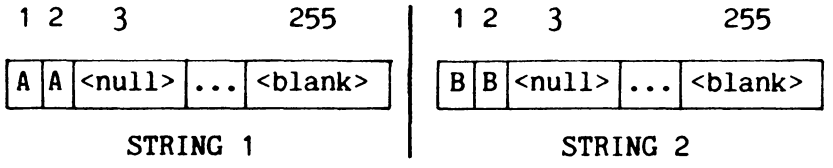
Figure 15: Stack Management

CHAPTER VIII

8. STRING MANAGEMENT

String management is controlled by a set of routines that simulate PL/I-like built in string functions. In order to implement the variable length string aspects of PL/I, character strings are delimited by a non-printable character. The lexical system uses the null character as the delimiter. The operations on these simulated varying length strings are performed by character functions which return string values whose maximum lengths are 255 characters. The resulting operation, therefore, produces a string 255 characters long in which the null character may be positioned anywhere from character zero to character 254. The result may in turn be assigned to a character field of a length more indicative of the particular process requiring the string function services.

The string "AA" would, in reality, be stored in a ten character field as "AA<null> ". If this string were concatenated with the string "B", the result would be "AAB<null> ... " with 251 blanks following the null character. However, if it was known that the actual concatenated string was never more than ten characters long, then the concatenation function could be assigned to a variable character field ten characters long (Figure 16). String functions are most important when defining literals, identifiers, comments, and numbers. These tokens are built character by character as the input is processed and, once completed, are passed back to the parser. Likewise, as each command token is entered, a command token string is built character by character and, once completed, passed back to the parser along with the token value.




```
CHARACTER*10 variable

variable = concat(string1,string2)
```

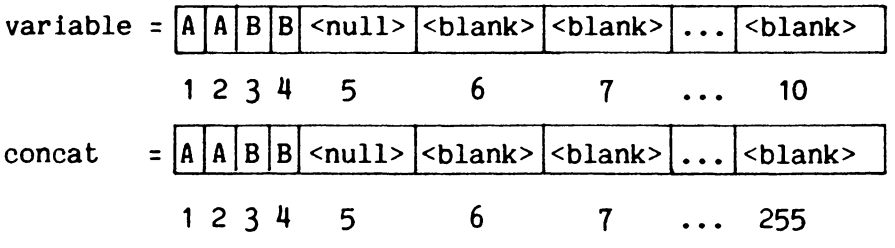


Figure 16: String Management

CHAPTER IX

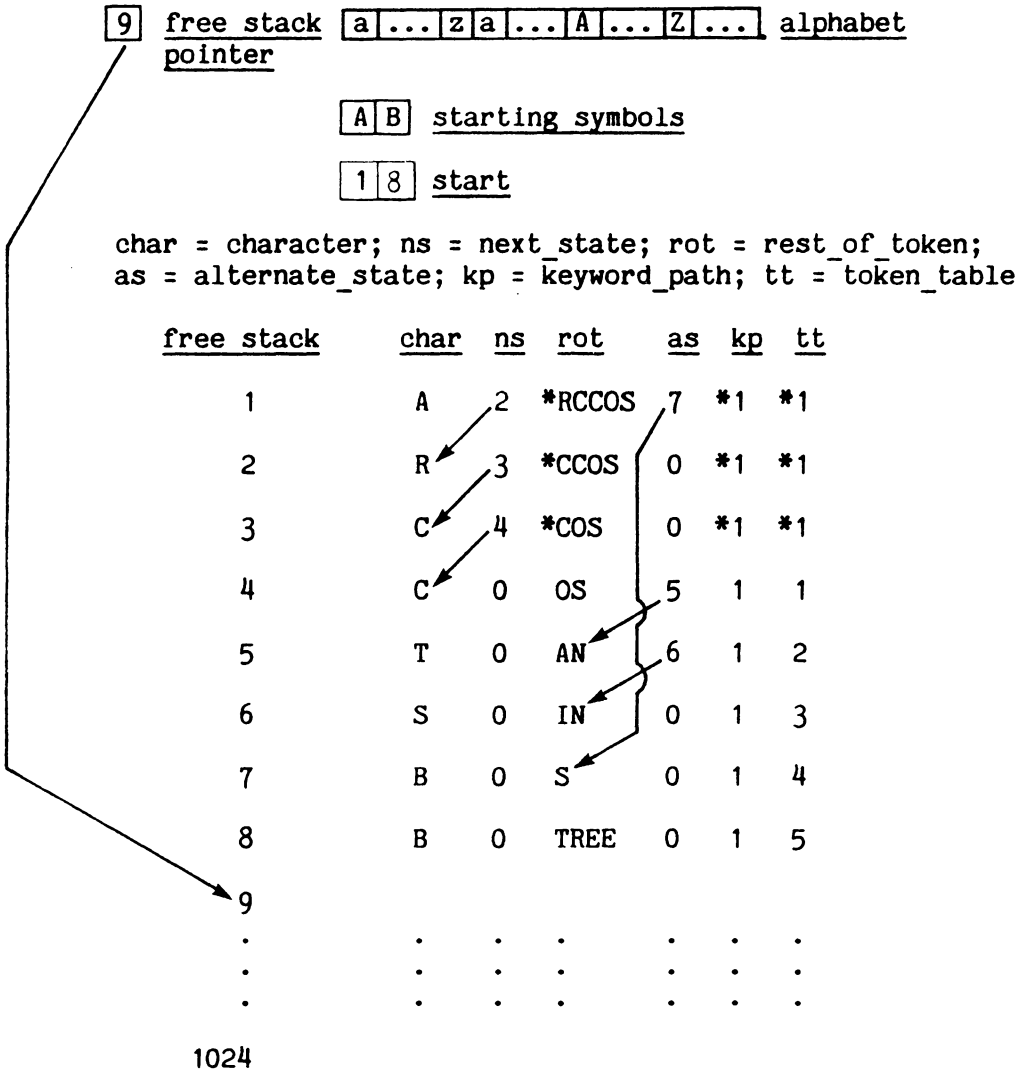
9. MAIN STORAGE

Main storage pertains to the lexical forest and consists of seven 1024 element arrays. Six of these arrays correspond to the six pieces of information stored at each node of the lexical forest. The seventh array monitors the pool of free nodes available to expand the lexical forest and is initialized by linking the elements of this array to form a linked stack. A variable pointer points to the top of this stack. When a new state must be added to the lexical forest, a node address is acquired by popping the stack of free nodes. This value is used as the index into the six other arrays to store the state information. Thus, the stack of free nodes is a pool of available indices into the other six arrays. As tokens are deleted from the lexical forest, the node indices are returned to the stack of free nodes.

The names and functions of the seven stacks are as follows:

1) CHARACTER - contains the transition character and holds 1 byte of information,
2) REST_OF_TOKEN - contains the concatenated recognizer path and holds up to thirty bytes of information,
3) NEXT_STATE - contains the link to the next level in the general tree and is an integer value,
4) ALTERNATE_STATE - contains the link to the alternate states at the same level and is an integer value,
5) TOKEN_TABLE - contains the token value or descriptor address and is an integer value, and
6) KEYWORD_PATH - contains the keyword value for the token and is an integer value.

Figure 17 is a schematic of main memory for a language containing five tokens. The names of each stack and pointer are the actual variable names used in the software.



* Values are meaningless

Language: ARCCOS, ARCSIN, ARCTAN, ABS, BTREE

Figure 17: Main Memory

CHAPTER X

10. FLOW CONTROL AND MISCELLANEOUS OPERATIONS

The processing of input is consistent. That is, regardless of the source of input (device independence), processing evokes the same sequence of procedural steps. The lexical system first initializes all data structures. This includes building the lexical forest, clearing the reverse lexical stacks, and linking main memory addresses. In short, the user's lexical environment is predefined at login time since all command, synonym, and noiseword tokens are loaded into the lexical forest. Finally, a prompt is displayed and the system waits for input.

At this point, the input may arrive from the I/O buffer, a VMS file, or the keyboard. In the module where the input is read, several validations are made. For example, if input is directed from the keyboard, token completion is turned off, an identifier is about to be defined, and the first input character is a carriage return, then a validation fails because the first character of an identifier cannot be a carriage return. Also, a blank validation exists. That is, if blanks between tokens are encountered, the tokens are processed solely within the I/O subsystem since these blanks do not interact with either the lexical forest nor the DFA table.

After the input character has passed all validations and if the input character is neither a blank between tokens nor a delete character, then a transition is attempted. If the transition succeeds, then the lexical

system pushes the related information onto the reverse lexical stacks, loops around, and waits for the next input character.

10.1 ECHOING TRANSITION CHARACTERS

If the input character passes all validations, then this character is immediately echoed to the output device. If the subsequent transition fails, the input character is deleted from the output device. This process may at first seem backwards. However, this ordering of procedural steps is necessary in order that there be a single output statement regardless of whether the processing mechanism is using the lexical forest or the DFA table. If the procedural steps were reversed, then there would have to be several output statements throughout the lexical system to echo the transition characters to the output device. Thus, the output mechanism is localized to a single output statement in the I/O subsystem.

10.2 ENDING LEXICAL ANALYSIS

There is a boolean variable that is initialized to a "false" value prior to beginning to accept input for a token. When the token has been recognized, this variable is set to a "true" value in order to end lexical analysis and to return the token string and token value to the parser.

10.3 SCREEN MANAGEMENT

There are several screen management routines that are executed before, during, and after lexical analysis that are transparent to the analysis operations. These are as follows:

1) If a token is not completed and the END_OF_LINE condition is encountered, then the partial token is removed from the current line and regenerated at the start of the next line,
2) If a command token ends in the last column of the line and token completing is off, the token is removed from the current line and regenerated at the start of the next line, and
3) If a literal, identifier, or comment is being defined and the user enters a carriage return rather than the ending delimiters, the ending delimiters will be generated and processed prior to processing the carriage return.

The first item involves popping the reverse lexical stacks until a null character is encountered in LINE and pushing the popped characters and a carriage return onto the I/O buffer. In this manner, the removed characters are regenerated as if they are once again being entered through the keyboard (Figure 18).

The second item involves definition command tokens. When these tokens are recognized, token completion is turned off in anticipation of the actual definition of a new token. These definition tokens are moved to the beginning of a new line to ensure that the definition token and the start of the definition are side by side on the same line. The definitions themselves may span several lines.

The third item simply involves pushing the ending delimiters and a carriage return onto the I/O buffer. Thus, once again, the automatically generated characters appear to the lexical system as if a user had entered them through a keyboard.

In summary, screen management is accomplished by simply pushing the proper sequence of characters onto the I/O buffer and allowing the software of the lexical system to take over.

```
I.  Column   1 2 3 ...              80

    Line 1   ^ ^ ^ ...      INTEGER_              SCREEN

    Line 2   <variable>
```

Before <variable> is entered, INTEGER_ is deleted and regenerated on line 2.

```
II. Column   1 2 3 ...              80

    Line 1   ^ ^ ^                  ^             SCREEN

    Line 2   INTEGER_<variable> ... ^
```

Note that after the definition command token is completed any further input evaluation uses the DFA table. Since leading blanks are not permitted in a variable or identifier an underscore is the final character of the command token to provide a visual separation between the command token and the definition.

Figure 18: Screen Management

10.4 COMMAND TOKEN CLASSES

There are three classes of command tokens. The first class is token completed. The second class consists of lexical endmarkers. These are tokens such as the operators "+", "-", etc. that can immediately follow a token such as a number. Lexical endmarkers are the final states in the DFA table. After the number, including the operator, is completed, the operator is deleted from the output device and regenerated via the I/O buffer and lexical forest. Thus, when the string "55.5+" is entered, the operator "+" will flash on the screen as it is deleted and regenerated. The third class involves token look-ahead. There are tokens that can have only one possible token follow as defined in the grammar rules. The most notable of these are the function tokens (e.g., SIN, TAN, etc.) that require a left parenthesis to follow. At the user's request, the lexical system will automatically generate the required parenthesis removing the responsibility from the user. This process is termed token look-ahead.

The justification for lexical end markers needs to be examined and is required because of two conflicting requirements. First, the grammar rules specify that tokens such as operators and numbers be parsed separately. Second, token completed tokens require an ending delimiter to prevent overtyping. This would require a blank between a number and operator. Unlike English-oriented command tokens, which are normally separated by a blank for readability, users do not necessarily separate numbers and operators by blanks. The lexical system allows the user to include or exclude the blank. If the blank is excluded, then the

operator serves as a final state in the DFA table and, at the same time, must serve as a token completed token to meet the requirements of the grammar rules. Thus, the operator will have to be regenerated. If a blank is included, then the blank serves as the final state in the DFA table and the operator is processed when it is entered with no regeneration required.

10.5 COMMENT BLOCKING

Comment blocking does not use the I/O buffer, but instead directly operates on the screen and reverse lexical stacks. There is a boolean variable that is set to a "true" value when a comment definition is completed; otherwise the variable is set to a "false" value. Also, the starting and ending positions of the comment are stored. If the next token is the beginning delimiter of a comment, and the boolean variable is set to a "true" value, then the previous token was also a comment. If, in addition, the starting position of the beginning comment delimiter is in column 1, then the beginning delimiter is advanced with blank fill until the delimiter aligns with the beginning comment delimiter on the previous line; otherwise, the "new" starting position is stored. If the position of the ending delimiter is less than the position of the ending delimiter of the previous comment, then this delimiter is advanced with blank fill until it aligns with the ending comment delimiter of the previous line; otherwise, the "new" ending position is stored.

CHAPTER XI

11. PERFORMANCE ISSUES AND MISCELLANEOUS PROBLEMS

Performance is an important aspect in any system. Fancier "user oriented" software may incur overhead. As a result, these newer systems are more dependent on a well-tuned computer hardware/software configuration. A well designed software system with greater overhead may perform less well than a simpler predecessor. The user community may show less acceptance of the new system despite the presumably more "easy-to-use" nature of the system because of these poor performance aspects. New software being developed on machines that do not experience heavy loads (e.g., VAX1) may perform very poorly in an operational environment under heavier loads. Because user acceptance may be linked in part to performance as well as "ease-of-use", these concepts should not be treated disjointly. That is, a designer should not claim that his software system is easy to use, and therefore will be widely accepted by the user community, without first studying the performance characteristics in a production environment. Note that performance is not being tied to the software alone, but rather, to the entire computer hardware/software configuration.

Before a project is undertaken or a software package purchased, research should be performed to determine if the current configuration can support the software with an acceptable response time (interactive system) or throughput (batch system). If the answer is no, then a further determination should be made to ascertain whether the

configuration can be modified such that the performance can be improved to an acceptable value without affecting the other systems and in a cost effective manner.

11.1 TOKEN COMPLETION AND PERFORMANCE PROBLEMS

The success of token completion and deletion may be tied to good performance characteristics. In order to take advantage of token completion, the user may wish to type recognizer characters as fast as he/she would type the entire token. If the user has to "wait" for token completion, there is no advantage over typing the entire token. A user who waits for token completion may be more "annoyed" than the user who types out the entire token and receives apparently immediate processing because the "slowness" of the user and the system are "matched".

If a user deletes a character and the system is slow at token deletion, then the user may strike the delete key a second time. When the system finally processes both deletions, the user may have to retype the characters that were inadvertantly deleted.

The discussion that follows concentrates on the software of the lexical system. The computer hardware configuration is unique for each installation and only the installations themselves know what the related performance aspects involve. The discussion that follows is speculative and simply forms the basis for future research.

11.2 TOKEN COMPLETION PERFORMANCE ASPECTS

The most expensive aspect of the lexical system is the initialization period when the lexical forest is built which occurs at the start of

each DBMS session. The required building time is a function of the number of tokens, the number of tokens that overlap on concatenated recognizer paths and the degree of overlap. The latter is critical because string operations must be invoked to split the concatenated recognizer path when an overlap exists with a token being added to the lexical forest.

Another performance aspect of token completion involves the traversal of internal recognizer paths versus concatenated recognizer paths. To complete an internal recognizer path, the nodes must be traversed and at each node, a single character is passed to an OS system service routine to perform the output operation. To complete a concatenated recognizer path, just the concatenated recognizer path must be passed to the OS service routine. Thus, there is no traversal of the lexical forest and only a single call is placed rather than one call per node.

A possible enhancement to the lexical forest would be to concatenate the internal recognizer characters of internal recognizer paths. Thus, the transition characters "E", " ", "C", "U", "R", "R", "E", "N", and "T" in Figure 19 would be concatenated. There is a trade off, however, between the speeds of token completion and the dynamic growth and shrinkage of the lexical forest. Remember that variables, relation names, etc. are added and deleted from the lexical forest as these variables are defined or deleted. In order to reconstruct the lexical forest when recognizer paths are concatenated requires potentially costly string operations. Although token completion might be faster, the definitions and deletions for variables may be slowed when using a dynamic allocation scheme. An alternate scheme is to dynamically

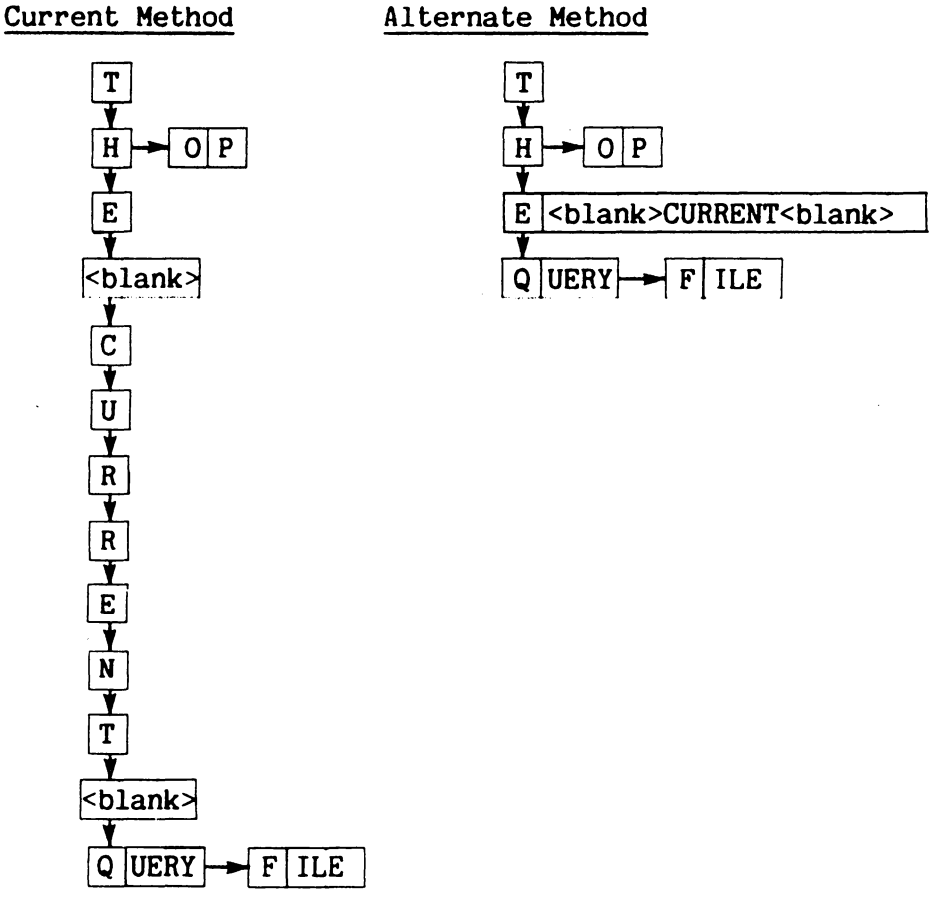


Figure 19: Enhancement to the Lexical Forest

allocate new tokens to the lexical forest, but to delete only logically. That is, when a deletion command is issued, the nodes are only marked as deleted. At some later time, a compaction system may be invoked to physically delete the marked nodes and to relink the entire forest. Compaction might be explicitly invoked by the user or when a particular percentage of available memory is in use, or during non-prime time. This would require dynamic memory allocation to reduce the probability of filling main memory. Currently, the static arrays used for main memory are only 1024 elements in length. A dynamic scheme would increase the size to the amount of virtual memory available to the process.

11.3 TOKEN OVERLAP

Another aspect of token completion involves the overlap of tokens. Complete tokens require a delimiter to prevent overtyping. The delimiter in the lexical system is the blank character. Suppose that the following two tokens are defined: "THE" and "THE CURRENT QUERY". In order to recognize the token "THE", three blanks must follow this token. The first blank corresponds to the blank in the token "THE CURRENT QUERY". The second blank corresponds to the character "C" in the token "THE CURRENT QUERY" and distinguishes the two tokens from each other. The third blank is the actual token delimiter (refer to 4.2 GRAPH-BASED DESCRIPTION). Another example involves the division operator, "/", and the beginning comment delimiter, "/*". Recall the rationale for the existence of lexical endmarkers. Clearly, the expression "VARIABLE1/VARIABLE2" is not possible. A blank must be entered after

the division operator to distinguish this token from the beginning comment delimiter. Thus, the user is forced to type "VARIABLE1/<blank>VARIABLE2". The blank is not a token delimiter since delimiters are not required in an expression. The blank is actually a recognizer state in the lexical forest. The goal, then, is to chose tokens such that these situations do not occur. The token "THE CURRENT QUERY" might be changed to "THE_CURRENT_QUERY" and a completely different set of comment delimiters might be defined.

11.4 HARD-CODED COMMAND TOKENS

Certain command tokens are hard-coded into the software. That is, the token values are referenced in the software. Therefore, if the token value is changed in the command file, this change must also be reflected in the software. Those tokens involved in token look-ahead or are automatically processed when a carriage return is depressed, are hard-coded. Again, this is a performance issue. Since token values are stored in the command file and are rarely changed once they are assigned, a decision was made to hard-code the token values to increase the speed of these processes. The alternative would be to search the lexical forest for the token to be processed in order to obtain the token value. Hard-coding causes a strange anomaly. If a synonym is defined for these tokens, any regeneration will cause the command token, not the synonym token, to be sent to the output device.

CHAPTER XII

12. FUTURE WORK

There are several modifications and areas of research that might be performed in the future. These include:

1. dynamic memory management,
2. an indexed error handling facility, and
3. performance evaluation studies.

12.1 DYNAMIC MEMORY MANAGEMENT

The use of dynamic memory management would require rewriting the lexical system in PL/I or PASCAL. The static memory arrays now used would be eliminated. The dynamic memory arrays would be limited by the virtual storage available to the process.

12.2 ERROR HANDLING FACILITY

The error handling facility currently contains a unique value for each routine that can generate an error message. Each value is a multiple of 100 (e.g., 100, 200, 300, etc.). A displacement for each message is added onto this value (e.g., first message = 101, second message = 102, etc). This number is passed to the error handling subroutine. The number is divided by 100 and the remainder is subtracted from the original value. This splits the single parameter back into the value corresponding to the routine where the error occurred and into the message displacement. The former number is used in a case statement to call the error message generating facility for the routine where the error occurred. The displacement is then used in another case statement to output the appropriate message.

An alternate scheme would be to use the error code as a hash key into an error file where a status code is stored for the set of actions that are to take place. The error file serves as an interface between the software of the lexical system and the error recovery procedures to be executed for each particular error. Thus, the error recovery mechanism can now be modified without requiring a recompilation of the lexical system. When an error occurs, the status code serves as a functional specification or a pointer to the proper recovery mechanism process. The set of status codes are to the error handling facility what the set of primitives are to command analysis or what token values are to token completion.

12.3 THREADED LEXICAL FORESTS

In a paper by [SIM84], the three pointers are eliminated by adding additional paths, or threads, to the lexical forest. The transition characters defining a recognizer path are concatenated into a single string which is stored at the first node of the recognizer path. Null characters are stored at the remaining nodes (Figure 20). There is a tradeoff in that the dynamic allocation and deletion of tokens into and out of the lexical forest is further complicated. However, token completion may be faster in that the threads permit the lexical system to "skip" nodes when traversing the forest. In addition, the software is simplified in that there is no longer a need to track three separate transitions with pointers.

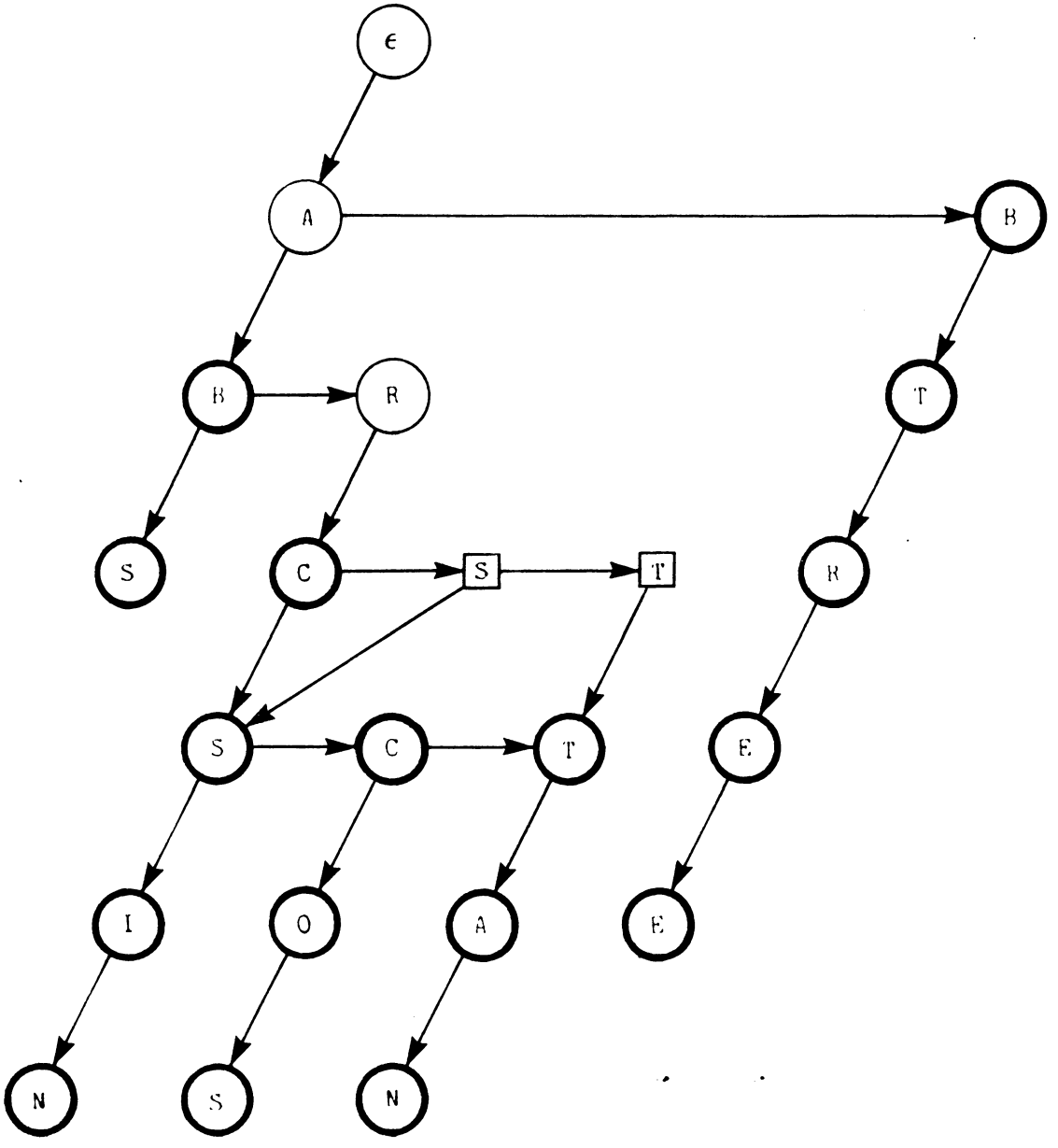


Figure 20: Mathematical Model of Token Completion

CHAPTER XIII

13. CONCLUSION

The lexical system is not intended to be a production system. This software is a research tool with which various user environments may be defined, studied, analyzed, and judged. The modification of user lexicons has, hopefully, been simplified. However, the work is far from complete; the system can still be improved. Software can be completed that permits the interactive modification of the command, synonym, and noiseword files. An LALR parser and an entire verified LALR grammar have been defined but not yet implemented. Various command procedures have been written to aid in software development and maintenance; some are not completed. An interactive documentor is partially developed. There is far more work than this young buck can tackle alone. Now is the time to pass the fruits of my labor to the next generation of students. However, I will remain in contact to help persue the endeavors and to hopefully see a fully operational system (I resisted from writing this way for eighty-seven pages; I could not resist ending the paper this way).

BIBLIOGRAPHY

[AHO79] Aho, Alfred V. and Jeffery D. Ullman, Principles of Compiler Design, Reading, Massachusetts, Addison-Wesley Publishing Company, 1979, 146-150.

[CARROL82] Carroll, John M., "The Adventure of Getting to Know a Computer," Computer, November 1982, 49-58.

[LEWIS78] Lewis II, P.M., et. al., Compiler Design Theory, Reading, Massachusette, Addison-Wesley Publishing Company, 1978, 67-83.

[REIS81] Reisner, Phyllis, "Formal Grammar and Human Factors Design of an Interactive Graphics System," IEEE Transactions On Software Engineering, Vol. SE-7, No. 2, March 1981, 229-240.

[SCREEN83] Denman, William F. Jr., Interview, Screen Play, Inc., Chapel Hill, North Carolina, 1983.

[SIM84] Simasek, Jon E., A Look at Some Language Design Tools and Their Use with Respect to the Dialogue Management System and Mini Database Management System, Department of Computer Science, Blacksburg, Virginia, Virginia Polytechnic Institute and State University, 1984.

[VATECH83] Anonymous, Department of Computer Science, MDB System Manual and User Guide, Blacksburg,Virginia, Virginia Polytechnic Institute and State University, 1983.

Appendix A

BNF Grammar for the Query Language of the MDB

```
%left 'or'
%left 'and'
%left 'not'
%nonassoc '<' '>' '=' '<=' '>=' '<>'
%left '+' '-'
%left '*' '/'
%left '**'
%right UNARY

%%
STATEMENTLIST  <----   STATEMENTLIST  S
STATEMENTLIST  <----   S
STATEMENTLIST  <----  'run'

S <----  'CHARACTERVAR'  '<=='           '?' ';'
S <----  'INTEGERVAR'    '<=='           '?' ';'
S <----  'REALVAR'       '<=='           '?' ';'
S <----  'BOOLEANVAR'    '<=='           '?' ';'
S <----  'CHARACTERVAR'  '<=='           ES ';'
S <----  'INTEGERVAR'    '<=='           E ';'
S <----  'REALVAR'       '<=='           E ';'
S <----  'BOOLEANVAR'    '<=='           EL ';'

S <----  'alias'         'IDENTIFIER'     ATTRIBUTENAME ';'
S <----  'alias'         'IDENTIFIER'    'BOOLEANVAR' ';'
S <----  'alias'         'IDENTIFIER'    'CHARACTERVAR' ';'
S <----  'alias'         'IDENTIFIER'    'INTEGERVAR' ';'
S <----  'alias'         'IDENTIFIER'    'REALVAR' ';'
S <----  'alias'         'IDENTIFIER'    RELATIONNAME ';'
S <----  'alias'         'IDENTIFIER'    'SCHEMANAME' ';'
S <----  'alias'         'IDENTIFIER'    'GROUPNAME' ';'

S <----  'character'     'IDENTIFIER' ';'
S <----  'character'     'IDENTIFIER'   '<=='          ES ';'
S <----  'character'     'IDENTIFIER'   '<=='          '?' ';'

S <----  'create'        'GROUPNAME'       'group' ';'
S <----  'create'        RELATIONNAME     'relation' ';'
S <----  'create'        'SCHEMANAME'      'schema' ';'

S <----  'delete'        'all'          'tuples' ';'
S <----  'delete'        'attribute'     ATTRIBUTENAME ';'
S <----  'delete'        'attribute'     ATTRIBUTENAME   'relation'     RELATIONNAME ';'
S <----  'delete'        'database'     'DATABASENAME' ';'
S <----  'delete'        'group'        'GROUPNAME' ';'
S <----  'delete'        'group'         'GROUPNAME'    'relation'     RELATIONNAME ';'
S <----  'delete'        'relation'     RELATIONNAME ';'
S <----  'delete'        'relation'     RELATIONNAME    'group'        'GROUPNAME' ';'
S <----  'delete'        'schema'       'SCHEMANAME' ';'
S <----  'delete'        'tuples' ';'
S <----  'delete'        'tuples'        'all' ';'
S <----  'delete'        'tuples'        'relation'     RELATIONNAME ';'
S <----  'delete'        'tuples'        'relation'     RELATIONNAME  'where'         EL ';'
S <----  'delete'        'tuples'        'where'         EL ';'
S <----  'delete'        'tuples'        'where'         EL          'relation'     RELATIONNAME ';'

S <----  'do'            STATEMENTLIST  'until'          EL           'enddo' ';'
S <----  'do'           'while'         EL              STATEMENTLIST 'enddo' ';'
S <----  'do'           'INTEGERVAR'    '<=='           DOLIST        STATEMENTLIST  'enddo' ';'
S <----  'do'           'REALVAR'       '<=='           DOLIST        STATEMENTLIST  'enddo' ';'
```

```
DOLIST          <----   E               'DELIMITER'     E
DOLIST          <----   E               'DELIMITER'     E               'DELIMITER'     E


S <----   MATCHEDIF ';'
S <----   UNMATCHEDIF ';'
MATCHEDIF       <----   'if'            EL              'then'          STATEMENTLIST   'else'          STATEMENTLIST   'endif'
UNMATCHEDIF     <----   'if'            EL              'then'          STATEMENTLIST   'endif'

S <----   'insert' ';'
S <----   'insert'        'tuples' ';'
S <----   'insert'        'tuples'        'relation'      RELATIONNAME ';'
S <----   'insert'        'attribute' ';'
S <----   'insert'        'attribute'     'relation'      RELATIONNAME ';'
S <----   'insert'        'relation'      RELATIONNAME ';'
S <----   'insert'        'relation'      RELATIONNAME    'group'         'GROUPNAME' ';'

S <----   'integer'       'IDENTIFIER' ';'
S <----   'integer'       'IDENTIFIER'    '<=='           E ';'
S <----   'integer'       'IDENTIFIER'    '<=='           '?' ';'

S <----   'load'          'group'         'GROUPNAME' ';'
S <----   'load'          'relation'      RELATIONNAME';'
S <----   'load'          'schema'        'SCHEMANAME' ';'
S <----   'load'          'relation'      RELATIONNAME    'group'         'GROUPNAME' ';'
S <----   'load'          'group'         'GROUPNAME'     'relation'      RELATIONNAME';'

S <----   'logical'       'IDENTIFIER' ';'
S <----   'logical'       'IDENTIFIER'    '<=='           EL ';'
S <----   'logical'       'IDENTIFIER'    '<=='           '?' ';'

S <----   'print' ';'
S <----   'print'         'blanklines' ';'
S <----   'print'         'group'         'GROUPNAME' ';'
S <----   'print'         E               'blanklines' ';'
S <----   'print'         FORMATLIST ';'
S <----   'print'         'page' ';'
S <----   'print'         'query'         'QUERYNAME' ';'
S <----   'print'         'schema'        'SCHEMANAME' ';'
S <----   'print'         'using'         'FORMATNAME' ';'

FORMATLIST      <----   FORMATLIST      'DELIMITER'     E
FORMATLIST      <----   FORMATLIST      'DELIMITER'     EL
FORMATLIST      <----   FORMATLIST      'DELIMITER'     ES
FORMATLIST      <----   FORMATLIST      'DELIMITER'     FORMATFUNCTION  E
FORMATLIST      <----   FORMATLIST      'DELIMITER'     FORMATFUNCTION  EL
FORMATLIST      <----   FORMATLIST      'DELIMITER'     FORMATFUNCTION  ES
FORMATLIST      <----   FORMATLIST      FORMATFUNCTION  E
FORMATLIST      <----   FORMATLIST      FORMATFUNCTION  EL
FORMATLIST      <----   FORMATLIST      FORMATFUNCTION  ES
FORMATLIST      <----   E
FORMATLIST      <----   EL
FORMATLIST      <----   ES
FORMATLIST      <----   FORMATFUNCTION  E
FORMATLIST      <----   FORMATFUNCTION  EL
FORMATLIST      <----   FORMATFUNCTION  ES

FORMATLIST      <----   FORMATLIST      'DELIMITER'     E       '#'             E
FORMATLIST      <----   FORMATLIST      'DELIMITER'     EL      '#'     E
FORMATLIST      <----   FORMATLIST      'DELIMITER'     EL      '!'     E
```

```
FORMATLIST      <----  FORMATLIST      'DELIMITER'     ES       '|'          E
FORMATLIST      <----  FORMATLIST      'DELIMITER'     FORMATFUNCTION   E          '#'          E
FORMATLIST      <----  FORMATLIST      'DELIMITER'     FORMATFUNCTION   EL         '#'          E
FORMATLIST      <----  FORMATLIST      'DELIMITER'     FORMATFUNCTION   EL         '|'          E
FORMATLIST      <----  FORMATLIST      'DELIMITER'     FORMATFUNCTION   ES         '|'          E
FORMATLIST      <----  FORMATLIST      FORMATFUNCTION  E          '#'           E
FORMATLIST      <----  FORMATLIST      FORMATFUNCTION  EL         '|'           E
FORMATLIST      <----  FORMATLIST      FORMATFUNCTION  EL         '#'           E
FORMATLIST      <----  FORMATLIST      FORMATFUNCTION  ES         '|'           E
FORMATLIST      <----  E          '#'           E
FORMATLIST      <----  EL         '#'           E
FORMATLIST      <----  EL         '|'           E
FORMATLIST      <----  ES         '|'           E
FORMATLIST      <----  FORMATFUNCTION  E          '#'           E
FORMATLIST      <----  FORMATFUNCTION  EL         '|'           E
FORMATLIST      <----  FORMATFUNCTION  EL         '#'           E
FORMATLIST      <----  FORMATFUNCTION  ES         '|'           E
FORMATFUNCTION <----  SPACINGFUNCTION
FORMATFUNCTION <----  SPACINGFUNCTION 'capitalize'
FORMATFUNCTION <----  SPACINGFUNCTION 'capitalize'     'underline'
FORMATFUNCTION <----  SPACINGFUNCTION 'underline'
FORMATFUNCTION <----  SPACINGFUNCTION 'underline'      'capitalize'
FORMATFUNCTION <----  'capitalize'
FORMATFUNCTION <----  'capitalize'     'underline'
FORMATFUNCTION <----  'underline'
FORMATFUNCTION <----  'underline'      'capitalize'

SPACINGFUNCTION <---- 'center'
SPACINGFUNCTION <---- 'nextto'
SPACINGFUNCTION <---- 'tab'
SPACINGFUNCTION <---- 'space'

S <----   'run'          'QUERYNAME' ';'
S <----   'run'          'QUERYNAME'     RUNPARALIST ';'

RUNPARALIST <---- RUNPARALIST       'DELIMITER'        E
RUNPARALIST <---- RUNPARALIST       'DELIMITER'        ES
RUNPARALIST <---- RUNPARALIST       'DELIMITER'        EL
RUNPARALIST <---- E
RUNPARALIST <---- ES
RUNPARALIST <---- EL

S <----   ES1            'select'        ATTRIBUTELIST    P1 ';'
S <----   ES1            'select'        ATTRIBUTELIST    P2 ';'
S <----   ES1            'select'        ATTRIBUTELIST    P3 ';'
S <----   ES1            'select'        ATTRIBUTELIST    P4 ';'
S <----   ES2            'select'        ATTRIBUTELIST    P1 ';'
S <----   ES2            'select'        ATTRIBUTELIST    P3 ';'
S <----   ES3            'select'        ATTRIBUTELIST    P1 ';'
S <----   ES3            'select'        ATTRIBUTELIST    P2 ';'
S <----   ES4            'select'        ATTRIBUTELIST    P1 ';'
S <----  'select'         ATTRIBUTELIST   P1 ';'
S <----  'select'         ATTRIBUTELIST   P2 ';'
S <----  'select'         ATTRIBUTELIST   P3 ';'
S <----  'select'         ATTRIBUTELIST   P4 ';'

ES1           <----   ES1           'select'        ATTRIBUTELIST    P5;
ES1           <----  'select'        ATTRIBUTELIST   P5;

ES2           <----   ES1           'select'        ATTRIBUTELIST    P6;
```

```
ES2          <----  ES2          'select'       ATTRIBUTELIST   P5;
ES2          <----  'select'      ATTRIBUTELIST  P6;

ES3          <----  ES1          'select'       ATTRIBUTELIST   P7;
ES3          <----  ES3          'select'       ATTRIBUTELIST   P5;
ES3          <----  'select'      ATTRIBUTELIST  P7;

ES4          <----  ES1          'select'       ATTRIBUTELIST   P8;
ES4          <----  ES2          'select'       ATTRIBUTELIST   P7;
ES4          <----  ES3          'select'       ATTRIBUTELIST   P6;
ES4          <----  ES4          'select'       ATTRIBUTELIST   P5;
ES4          <----  'select'      ATTRIBUTELIST  P8;

P1           <----
P1           <----  'relation'    RELATIONNAME
P1           <----  'relation'    RELATIONNAME   'where'        EL
P1           <----  'where'       EL

P2           <----  'relation'    RELATIONNAME   SORTLIST
P2           <----  'relation'    RELATIONNAME   SORTLIST       'where'        EL
P2           <----  SORTLIST
P2           <----  SORTLIST      'relation'     RELATIONNAME
P2           <----  SORTLIST      'relation'     RELATIONNAME   'where'        EL
P2           <----  SORTLIST      'where'        EL

P3           <----  'relation'    RELATIONNAME   'unique'
P3           <----  'relation'    RELATIONNAME   'unique'       'where'        EL
P3           <----  'unique'
P3           <----  'unique'      'relation'     RELATIONNAME
P3           <----  'unique'      'relation'     RELATIONNAME   'where'        EL
P3           <----  'unique'      'where'        EL

P4           <----  'relation'    RELATIONNAME   SORTLIST       'unique'
P4           <----  'relation'    RELATIONNAME   SORTLIST       'unique'       'where'        EL
P4           <----  SORTLIST      'unique'
P4           <----  SORTLIST      'relation'     RELATIONNAME   'unique'
P4           <----  SORTLIST      'relation'     RELATIONNAME   'unique'       'where'        EL
P4           <----  SORTLIST      'unique'       'relation'     RELATIONNAME
P4           <----  SORTLIST      'unique'       'relation'     RELATIONNAME   'where'        EL
P4           <----  SORTLIST      'unique'       'where'        EL
P4           <----  'unique'      'relation'     RELATIONNAME   SORTLIST
P4           <----  'unique'      'relation'     RELATIONNAME   SORTLIST       'where'        EL
P4           <----  'unique'      SORTLIST
P4           <----  'unique'      SORTLIST       'relation'     RELATIONNAME
P4           <----  'unique'      SORTLIST       'relation'     RELATIONNAME   'where'        EL
P4           <----  'unique'      SORTLIST       'where'        EL

P5           <----  'relation'    RELATIONNAME   'where'
P5           <----  'where'

P6           <----  'relation'    RELATIONNAME   SORTLIST       'where'
P6           <----  SORTLIST      'relation'     RELATIONNAME   'where'
P6           <----  SORTLIST      'where'

P7           <----  'relation'    RELATIONNAME   'unique'       'where'
P7           <----  'unique'      'relation'     RELATIONNAME   'where'
P7           <----  'unique'      'where'

P8           <----  'relation'    RELATIONNAME   SORTLIST       'unique'       'where'
P8           <----  'relation'    RELATIONNAME   'unique'       SORTLIST       'where'
```

```
P8              <----   SORTLIST         'relation'      RELATIONNAME 'unique'       'where'
P8              <----   SORTLIST        'unique'         'relation'      RELATIONNAME'where'
P8              <----   SORTLIST        'unique'        'where'
P8              <----  'unique'          'relation'      RELATIONNAME  SORTLIST       'where'
P8              <----  'unique'          SORTLIST        'relation'      RELATIONNAME'where'
P8              <----  'unique'          SORTLIST       'where'

SORTLIST        <----  'ascendingon'    ATTRIBUTENAME
SORTLIST        <----   SORTLIST        'DELIMITER'     'ascendingon'   ATTRIBUTENAME
SORTLIST        <----  'descendingon'   ATTRIBUTENAME
SORTLIST        <----   SORTLIST       'DELIMITER'     'descendingon'  ATTRIBUTENAME


S <----  'set' ';'

S <----  'store'          'group'         'GROUPNAME' ';'
S <----  'store'          'query'         'QUERYNAME' ';'
S <----  'store'          'relation'      'REALATIONNAME' ';'
S <----  'store'          'schema'        'SCHEMANAME' ';'

S <----  'switch'         E               ECASELIST       'endswitch' ';'
S <----  'switch'         ES              ESCASELIST      'endswitch' ';'
S <----  'switch'         STATEMENTLIST   E               ECASELIST       'endswitch' ';'
S <----  'switch'         STATEMENTLIST   ES              ESCASELIST      'endswitch' ';'

ECASELIST       <----  'case'          'integer'          ':'              STATEMENTLIST
ECASELIST       <----  'case'          'number'          ':'              STATEMENTLIST
ECASELIST       <----  'case'          'REALVAR'       ':'              STATEMENTLIST
ECASELIST       <----  ECASELIST       'case'          'integer'     ':'          STATEMENTLIST
ECASELIST       <----  ECASELIST       'case'          'number'      ':'          STATEMENTLIST
ECASELIST       <----  ECASELIST       'case'          'INTEGERVAR'          ':'          STATEMENTLIST
ECASELIST       <----  ECASELIST       'case'          'REALVAR'     ':'          STATEMENTLIST

ESCASELIST      <----  'case'                      'CHARACTERVAR'                    ':'          STATEMENTLIST
ESCASELIST      <----  'case'                      'LITERAL'                    ':'          STATEMENTLIST
ESCASELIST      <----  ESCASELIST     'case'      'CHARACTERVAR'          ':'          STATEMENTLIST
ESCASELIST      <----  ESCASELIST     'case'      'LITERAL'          ':'          STATEMENTLIST

S <----  'real'          'IDENTIFIER'       '<=='          E ';'
S <----  'real'          'IDENTIFIER' ';'
S <----  'real'          'IDENTIFIER'       '<=='          '?' ';'

S <----  'update' ';'
S <----  'update'          ATTRIBUTELIST ';'
S <----  'update'          ATTRIBUTELIST   'relation'      RELATIONNAME ';'
S <----  'update'          ATTRIBUTELIST   'relation'      RELATIONNAME  'where'          EL ';'
S <----  'update'          ATTRIBUTELIST   'where'          EL ';'
S <----  'update'          ATTRIBUTELIST   'where'          EL          'relation'      RELATIONNAME ';'

S <----  '/*'             'CHARSTRING' ';'

ATTRIBUTELIST  <----  'all'           'attributes'
ATTRIBUTELIST  <----   ATTRIBUTENAME
ATTRIBUTELIST  <----   ATTRIBUTELIST 'DELIMITER'       ATTRIBUTENAME

ATTRIBUTENAME    <----  RELATIONNAME    SATTRIBUTENAME
ATTRIBUTENAME    <----  SATTRIBUTENAME

SATTRIBUTENAME  <----   'BATTRIBUTENAME'
SATTRIBUTENAME  <----   'CATTRIBUTENAME'
```

```
SATTRIBUTENAME  <----   'IATTRIBUTENAME'
SATTRIBUTENAME  <----   'NATTRIBUTENAME'

EL              <----   EL              'or'            EL
EL              <----   EL              'and'           EL
EL              <----  'BATTRIBUTENAME'
EL              <----  'BATTRIBUTENAME' '('             E               ')'
EL              <----  'BOOLEANVAR'
EL              <----   NRANGEPRED
EL              <----   SRANGEPRED
EL              <----   'true'
EL              <----   'false'
EL              <----  'not'             EL
EL              <----  'verify'          '('             ES              'DELIMITER'     ES              ')'
EL              <----  '('               EL              ')'

ELIST           <----   ELIST           'DELIMITER'     E
ELIST           <----   E

ES              <----   'LITERAL'
ES              <----  'CATTRIBUTENAME'
ES              <----  'CATTRIBUTENAME' '('             E               ')'
ES              <----  'CHARACTERVAR'
ES              <----  MAXMINFUNCTION   '('             ESLIST          ')'
ES              <----  'substring'      '('             ES              'DELIMITER'     E               'DELIMITER'  E          ')'
ES              <----  'concatenate'    '('             ESLIST          ')'

ESLIST          <----   ESLIST          'DELIMITER'     ES
ESLIST          <----   ES

E               <----   E               '+'             E
E               <----   E               '-'             E
E               <----   E               '*'             E
E               <----   E               '/'             E
E               <----   E               '**'            E
E               <----   EFUNCTIONE1     '('             E               ')'
E               <----   'length'        '('             ES              ')'
E               <----   'mean'          '('             ELIST           ')'
E               <----  'count'          '('            'attribute'      'DELIMITER'      ATTRIBUTENAME    ')'
E               <----  'count'          '('            ATTRIBUTENAME    'DELIMITER'      'attribute'      ')'
E               <----  'count'          '('            'relation'       'DELIMITER'      RELATIONNAME     ')'
E               <----  'count'          '('            RELATIONNAME     'DELIMITER'      'relation'       ')'
E               <----  'index'          '('             ES              'DELIMITER'      ES              ')'
E               <----  'IATTRIBUTENAME'
E               <----  'NATTRIBUTENAME'
E               <----  'IATTRIBUTENAME' '('             E               ')'
E               <----  'NATTRIBUTENAME' '('             E               ')'
E               <----  'mod'            '('             E               'DELIMITER'     E               ')'
E               <----  'round'    '('          E               'DELIMITER'     E               ')'
E               <----   'NUMBER';
E               <----   'standard deviation'          '('             ELIST           ')'
E               <----  'sum'          '('            'attribute'      'DELIMITER'      ATTRIBUTENAME    ')'
E               <----  'sum'          '('            ATTRIBUTENAME    'DELIMITER'      'attribute'      ')'
E               <----  'sum'          '('            'relation'       'DELIMITER'      RELATIONNAME     ')'
E               <----  'sum'          '('            RELATIONNAME     'DELIMITER'      'relation'       ')'
E               <----  'truncate'     '('             E               'DELIMITER'      E               ')'
E               <----   'variance'          '('             ELIST           ')'
E               <----  '('               E               ')'
E               <----  '-'               E               {%prec UNARY}
```

```
EFUNCTIONE1     <----  'abs'
EFUNCTIONE1     <----  'arccos'
EFUNCTIONE1     <----  'arcsin'
EFUNCTIONE1     <----  'arctan'
EFUNCTIONE1     <----  'cos'
EFUNCTIONE1     <----  'e'
EFUNCTIONE1     <----  'ln'
EFUNCTIONE1     <----  'log'
EFUNCTIONE1     <----  'sin'
EFUNCTIONE1     <----  'tan'

MAXMINFUNCTION  <----  'max'
MAXMINFUNCTION  <----  'min'

NRANGEPRED      <----  NRANGEPRED       RELOP          E
NRANGEPRED      <----  E                RELOP          E

SRANGEPRED      <----  SRANGEPRED       RELOP          ES
SRANGEPRED      <----  ES               RELOP          ES

RELATIONNAME    <----  'GROUPNAME'      '.'            'RELATIONNAME'
RELATIONNAME    <----  'RELATIONNAME'

RELOP           <----  '<'
RELOP           <----  '>'
RELOP           <----  '='
RELOP           <----  '<>'
RELOP           <----  '<='
RELOP           <----  '>='
```

Appendix B

Sample FORTRAN coding of the Lexical System

```
 SUBROUTINE lex(command_completion_off,complete,token_type,
$           idstring,status_code,yes_blkcoment,query_from_file,
$           file_name)

 CHARACTER*1 temp_character
 CHARACTER*2 input_character
 CHARACTER*(*) file_name,idstring

 INTEGER current_node,dfa_code,endnode,finish_state,
$        ignorant_pointer,knowledge_pointer,newcurnod,
$        next_node,status_code,token_type

 LOGICAL command_completion_off,comment_newline,complete,
$        end_comment,force_blank,move_comment,
$        partial_cmd_completion,query_from_file,yes_blkcoment

 INCLUDE 'constants.for'
 INCLUDE 'functions.for'
 INCLUDE 'lextables.for'
 INCLUDE 'lexstruct.for'

 DATA finish_state/0/,force_blank/.false./,move_comment/.false./,
$     end_comment/.false./,dfa_code/0/

 CALL prelex(status_code,complete,current_node,idstring,
$     dfa_code,input_character,partial_cmd_completion,
$     command_completion_off,ignorant_pointer,knowledge_pointer)
 DO WHILE(.not.complete)

    current_node=topint(lexpath)
    command_completion_off=toplog(cmd_completion_off_stack)
    input_character=nextchar(command_completion_off,status_code,
$                   complete,dfa_code,current_node,next_node,
$                   idstring,force_blank,query_from_file,file_name,
$                   comment_newline,yes_blkcoment,
$                   partial_cmd_completion)
    temp_character=substr(input_character,1,1)
    IF (temp_character.eq.delete)
$       THEN

          CALL precmddel(idstring,temp_character,
$              partial_cmd_completion)
          CALL delclenup(status_code,dfa_code,move_comment,
$              force_blank,end_comment,temp_character,
$              ignorant_pointer,knowledge_pointer,next_node,
$              command_completion_off,complete)

       ELSE

          call_dellex=false
          IF ((command_completion_off).and.
$             ( line_position.eq.1))
$             current_node=newcurnod(dfa_code,idstring,
$                          current_node)
          IF (command_completion_off)
$             THEN

                CALL prepnone(status_code,input_character,dfa_code,
$                    complete,idstring,command_completion_off,
$                    partial_cmd_completion,current_node,next_node)
```

```
            ELSE

               CALL prepcmplt(current_node,input_character,
$                   complete,force_blank,query_from_file,
$                   file_name,partial_cmd_completion,next_node,
$                   command_completion_off,ignorant_pointer,
$                   knowledge_pointer)

          ENDIF

    ENDIF

 ENDDO
 CALL postlex(token_type,command_completion_off,current_node,
$     force_blank,next_node,yes_blkcoment,move_comment,dfa_code,
$     idstring,end_comment,comment_newline,query_from_file)

 END
```

The vita has been removed from the scanned document