THIS BOOK CONTAINS NUMEROUS PAGES WITH THE ORIGINAL PRINTING ON THE PAGE BEING CROOKED.

THIS IS THE BEST IMAGE AVAILABLE.

THIS BOOK CONTAINS NUMEROUS PAGES WITH THE PAGE NUMBERS CUT OFF

THIS IS AS RECEIVED FROM THE CUSTOMER

EXTENSIBILITY OF LEXICAL ANALYZERS

by

GARY ANDERSON

B. S., Kansas State University, 1972

A MASTERS REPORT

submitted in partial fulfillment of

the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas
1973

Approved by:

Virgil E. Wallentine

Major Professor

TABLE OF CONTENTS

TABLE OF CONTENTS CONTINUED

TABLE OF CONTENTS CONTINUED

LIST OF FIGURES

LIST OF FIGURES CONTINUED

LIST OF TABLES

ACKNOWLEDGEMENT

The author wishes to express his sincere thanks and appreciation to his committee, Dr. Virgil E. Wallentine, Assistant Professor of Computer Science at Kansas State University, Dr. Paul S. Fisher, Head of the Department of Computer Science at Kansas State University, and Dr. William J. Hankley, Associate Professor of Computer Science at Kansas State University, for their invaluable guidance, direction, and assistance in all phases of the preparation of this paper. A note of thanks to David Gries of Cornell University whose book "Compiler Construction for Digital Computers" served as a guide for the finite-state lexical analyzers used in this paper. Finally, special thanks to fellow students in the Department of Computer Science at Kansas State University for their many hours of help in preparing and editing this paper.

It is acknowledged that some of the basic research for this paper was accomplished while the author was employed to design a lexical analyzer for the language EZE as a Graduate Research Assistant for the Department of Computer Science at Kansas State University.

ABSTRACT

A tutorial report dealing with processors which select grammar primitives from a source language. The processors presented range from the most basic finite-state machines up to the most sophisticated analyzers capable of inserting synonyms for changing the meaning of language symbols and inserting new language symbols. The analyzers discussed are based on the theory of finite-state machines and the proposal of adding semantic routines to these machines.

Two analyzer systems are presented. The first processor is AED-RWORD, "the industry standard", which was developed at M.I.T. as part of the AED-1 language translator. The RWORD system automatically generates lexical processors from descriptions of the language symbols given in terms of regular expressions. The second system presented is a lexical analyzer hand-coded from the proposed GENSCAN concepts. The GENSCAN concepts are a set of guidelines based on the theory of inserting semantic routines into a finite-state machine.

It is proposed that functions exist which allow the dynamic changing of the tables in such a manner as to change the meaning of symbols in the language being processed. These functions may be applied to either the RWORD lexical processor or to the processor constructed by GENSCAN concepts. These functions allow the language user the option of changing the meaning of symbols in order that the language is more meaningful to that individual.

I. INTRODUCTION

BACKGROUND

The function of a lexical analyzer is to read the original source program characters and assemble them into meaningful symbols, sometimes called tokens. The extensible lexical analyzers proposed in this paper are table-driven finite-state machines as opposed to less automatic ad hoc systems. A finite-state lexical analyzer's transition from its current state to its next state is driven by character classes represented as tables of regular expressions. Regular expressions allow automatic construction of the analyzer and make proof of validity of the automaton feasible. A table-driven machine is used in order to allow for static, interactive and dynamic extensibility of the lexical analyzer. A finite-state lexical analyzer may be constructed by either of the following two methods:

1. A constructor can be programmed which allows input of regular expressions for classes of symbols and outputs a lexical analyzer (AED-RWORD) [6, 9, 10].
2. A modular scanning algorithm may be defined which is sufficiently generalized to cover most uses for a lexical analyzer (GENSCAN).

The first approach requires a very large, complicated regular expression processor which builds a deterministic finite-state machine from a library of standard procedures. An automatic constructor removes the burden of program logic from the designer of the lexical analyzer, but the use of standardized modules tends to make the lexical analyzer somewhat inefficient. The second technique is a structuring extension

of an algorithm described by Gries [5]. The proposed algorithm is structured in such a way that if a portion of the lexical analyzer is not needed, then it may be omitted, allowing a more compact code for the scanner. In the extreme cases, when the algorithm does not cover a situation, the user may write a procedure to complement the general algorithm. The modularly-structured approach to building a finite-state lexical analyzer does place some of the burden of program logic on the designer; and the predefined modules do make the machine less efficient than a hand-coded lexical analyzer. However, the modular approach does not require the definition of the regular expression processor and automatic constructor.

EXTENSIBILITY

The dynamic nature of the tables used by the lexical analyzer provides the machine's extensibility. An extensible lexical analyzer has three types of tables: First, the tables which describe the class of characters (class tables). Next, a set of tables used to look up symbols to find their internal code (look-up tables), and last, a set of tables used to store synonyms for symbols used in the look-up tables (synonym tables). An efficient method of language development is to define a basic subset of the language and expand from this base. Allowing the lexical analyzer to expand along with the language, without completely rewriting the analyzer, can be accomplished by a proposed set of functions which allow for dynamic changing of class and lookup tables. When specific rules are followed as to the content of classes

and lookup tables, these functions are capable of checking the validity of new characters and symbols to the language. Synonym tables are dynamic in the sense that they are changeable by the source program being interpreted. These tables allow the source language user the option of changing the definition of keywords, delimiters or operators during execution of his program. This redefinition of symbols is carried out by what is referred to as the "Henceforth" function, which gives the new symbol the same internal code as the old symbol. One of the problems with most modern programming languages is that the vocabulary is oriented towards one user community. It is proposed that a redefinition of the vocabulary may be accomplished by the user through the function described above. Functions which allow the user to change the tables do degrade the efficiency of the analyzer. However, the tokens passed to the rest of the translator remain the same; therefore, no degradation of the rest of the system takes place. These features give the designer and user a very flexible lexical analyzer which is easy to design and extremely dynamic in character.

DEVELOPMENT

The approach developed in this paper is to first discuss reasons for separating the lexical and syntax analysis, followed by the GENSCAN approach to defining a scanning algorithm. The uses and methods of incorporating synonyms and extending tables followed by a review of the extensibility problem and the alternatives to the proposed approaches are given. For the reader who is not familiar with automata theory,

appendix A is a tutorial on the relationship of lexical analyzers to finite-state machines. Appendix B contains a discussion of the "industry standard" AED-RWORD approach to defining a scanning algorithm. The main portion of the report concludes with a comparison of the GENSCAN and RWORD approaches to building a scanning algorithm with consideration towards efficiency and extensibility.

II. SEPARATION OF LEXICAL AND SYNTACTIC ANALYSIS

JUSTIFICATION OF THE LEXICAL ANALYZER

With some exceptions, lexical properties have been assigned a very minor role in computer languages and lexical processing has been incorperated as an incidental part of the syntactic analysis program. It is often justly asked why the lexical analysis cannot be incorperated into the syntactic analysis.

Several good reasons for separating lexical from syntactical analysis are given here.

1. A large portion of compile-time is spent in scanning characters. Separation allows one to concentrate on reducing this time.
2. The syntax of symbols can be described by very simple grammars. If scanning is separated from syntax recognition, the development of efficient parsing techniques which are particularly well-suited for these grammars is possible, moreover the development of automatic methods for constructing scanners which use these efficient techniques is possible.
3. Since the scanner returns a symbol instead of a character, the syntax analyzer actually gets more information about what to do at each step.
4. Development of high-level languages requires attention to both lexical and syntactic properties. Separation of the two allows us to investigate them independently.
5. Often one has two different hardware representations for the same

language. Separation allows writing of one syntactic analyzer and several scanners (which are simpler and easier to write), one for each source program representation and/or input device. Each scanner translates the symbols into the same internal form used by the syntactic analyzer [5].

III. GENSCAN

INTRODUCTION TO GENSCAN

GENSCAN is not itself a lexical analyzer, but a set of concepts of how to apply semantic rules to a finite-state machine in order to produce almost any lexical analyzer needed by the designer. GENSCAN (Generalized Scanner constructor)consists of a set of concepts for covering context decisions and methods for building tables which allow the resulting scanner to be extensible. The GENSCAN approach is that the translator designer defines his own lexical analyzer, class, and look-up tables from the concepts presented in this paper. Then the tables may be expanded or contracted as needed by the designer or the user of the translator by calling functions which make up the table constructor. Figure 1 is a diagram of the proposed structure of a lexical analyzer built with GENSCAN concepts.

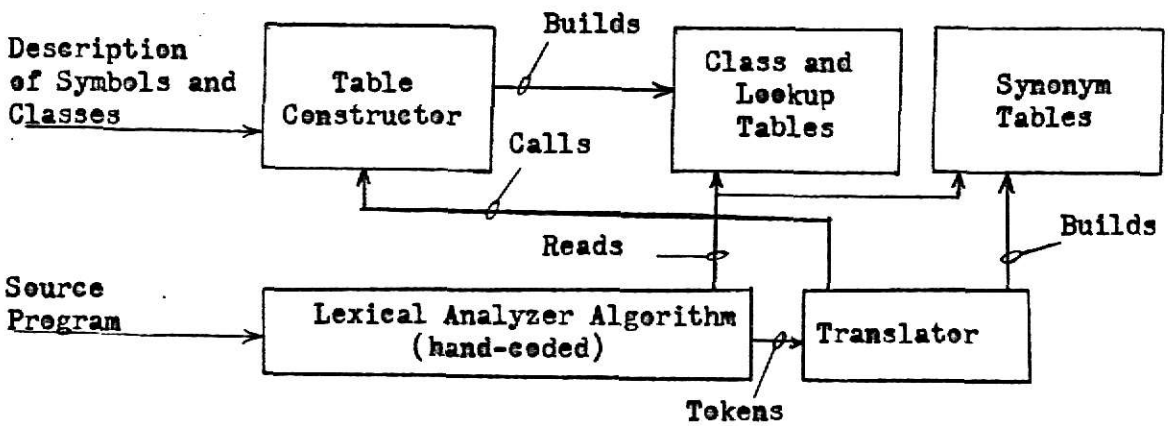


Figure 1. The basic GENSCAN System.

In order to discuss a generalized system, a general set of language

features must be described. The following list of primitives is given to describe the features of a general lexical analysis language. Its features will be used in examples throughout the rest of the paper.

I. Identifiers

A. Variable Names: A A3B A-B D0I0I

B. Unquoted Keywords: WRITE BEGIN DO BIN

C. Quoted Keywords or Reserved Words: "BEGIN" .EQ.

II. Constants

A. Integers: 23 0

B. Decimals: 2.3 2.

C. Scientific Notation: .23E5 1E-7 5.E2

D. Special Digits: 38A"HEX" 255"OCTAL" 10101"BIN"

III. Delimiters

A. Single Delimiters

1. Operators: + * = / ← ↑

2. Grouping Characters: ( ) [ ] < >

3. Separators: ; , : .

B. Double Delimiters

1. Operators: /* ¬= // <=

2. Grouping Characters: <<>> " "

3. Separators: :: ..

IV. Literals

A. Strings: 'CHARACTER STRING' '1101'

V. Skips

A. Ignore: "carriage return"

B. Spacer or Invtermin: "blank"

C. Comment: /*COMMENT*/ /*2.3*/

REGULAR GRAMMARS VS. CONTEXT-DEPENDENT GRAMMARS

The question may arise as to the reason that the lexical analyzer should handle context-dependent situations when they can be handled by the syntactic analyzer. It is true that the lexical analyzer does not have to parse complex structures, but there are several good reasons why it should.

1. If the scanner is able to efficiently parse these structures without a loss of semantics, then it does not matter where the parsing takes place.
2. Allowing the scanner to relieve some of the burden of the syntax parser results in a simpler syntactical analyzer which allows the development of a more efficient and automatic syntactical parsing algorithm.
3. If the lexical analyzer is allowed to parse some of the complex structures found in the source grammar, partial extensibility of the language can be accomplished in an efficient manner and with minimal or no change to the rest of the translator.

GENSCAN SEMANTICS

The lexical analyzer constructed from the GENSCAN concepts is a finite-state machine (fa) which follows the rules set down in the sections on automata theory. As long as the grammar being parsed is a regular grammar, the basic fa is able to parse the language. It is when the grammar contains more complex structures that the fa fails. In order for the fa to parse these complex structures, semantic rules are

added to the fa in the form of function calls. The semantic insertions used in GENSCAN are similar to those described in Appendix A's section on Lexical Semantics. The following is a list of semantic routines used by GENSCAN:

1. GETCHAR is a function which retrieves the next character of the source text, classifies it if it is in a class,and returns.
2. ADD is a function which concatenates the current character to the string of characters already in the current symbol and returns.
3. LOOKUP is a function which searches the table indicated for the current symbol and returns its internal code if the symbol is found. Otherwise a zero is returned.
4. ERROR is a function which writes an error message that an undefined character is encountered and returns.
5. MARK is function which marks the present character in the source text and in the current symbol so that backup can be accomplished and returns.
6. BACKUP is a function which moves the source location pointer used by GETCHAR, back to the last character which was marked in the source text, removes all characters in the current symbol back to the last character which was marked,and returns.

EXTENSIBILITY

The proposed answer to the question "What makes a lexical analyzor extensibile?" is:

1. A dynamic scanning algorithm or a scanning algorithm which is gen-

erally enough to cover all known cases.

2. Dynamic tables.
   A. The ability to build synonym tables for symbols in lookup tables.
   B. The ability to add and delete symbols represented in the lookup tables.
   C. The ability to add and delete characters representing directed arcs in the state diagram.
   D. The ability to build synonym tables for characters representing directed arcs.

At the present time there is not a dynamic lexical analyzer, but with some of the concepts presented here and with more research, such an algorithm seems feasible. It is feasible that the lexical analyzer may be capable of dynamically changing itself by selecting new machines from a library of modules. Therefore, if the designer of a lexical scanner wants a processor which is capable of extensibility, he must design an algorithm which is capable of handling all known cases.

The SPEAKEASY language [2, 3] uses a statement called HENCEFORTH which allows the language user the option of adding synonyms for keywords, or delimiters/operators in the language. This function allows the user to change the meaning of symbols so they are more oriented toward the jargon used in that users field. The proposal for adding this feature is presented in the section on LANGUAGE SYNONYMS. The ability to add and delete symbols in the language has two evident uses. First,

the designer of the language can start with a basic subset of his language and, by adding symbols, he can extend that language at his leisure. The second implication is that if a language is to be extensible, the user must be able to add operators and keywords when new features are inserted in the language. The dynamic changing of tables which allows these features is accomplished by functions discussed in the section DYNAMIC TRANSITIONS.

BASIC LEXICAL CONFIGURATIONS

The configurations that arise when analyzing a fa used as a lexical analyzer are of three types. The first configuration being that a character or class of characters, representing a transition in the fa, is independent of all other transitions in the fa. That is, the character or class of characters does not need to be unique and may contain characters used elsewhere in the fa. An example is that the class of characters representing all but the first character in an identifier may have characters in common with other language symbols. The character or class of characters that arises in this situation is called independent. The second configuration is when a character or class of characters must be unique from other characters used as transitions in the fa. An example being that the starting character of an identifier must be unique from the character used to describe the starting charac- of other symbols. This group of characters or class of characters is called unique and is said to be mutually exclusive of other classes. The third configuration arises when two or more classes of characters

have several characters in common. This situation arises when describing the class of characters which make up single delimiters and the class which describes the starting characters of double delimiters. This type of class is called nonunique and is said to have characters in common with other classes.

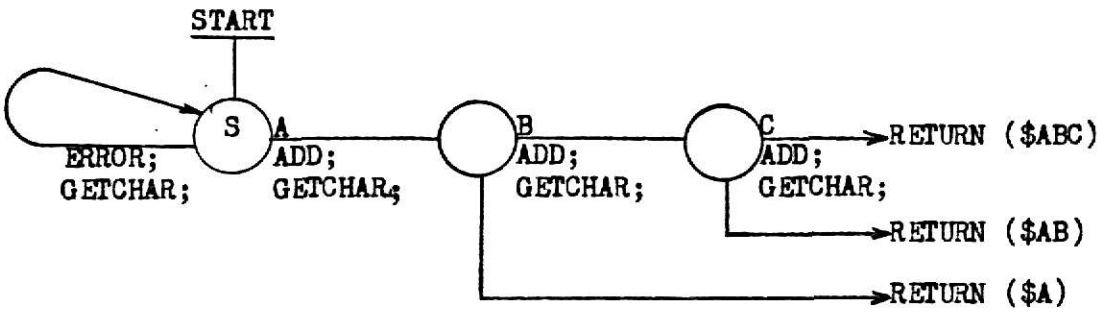
With the above configurations in mind, the designer can produce a fa lexical analyzer which will parse most language primitives with no problems. Two problems arise when trying to make changes in the transitions on previously designed machines. The first problem is that of trying to add a symbol which causes a unique class to become nonunique. An example being that of trying to add the dollar sign as a starting character for an identifier when it is also used to start some other symbol. In this case the user is trying to change the characters themselves. The second problem arises when a total symbol is to be added to a primitive group. In this situation the new symbol may cause a unique class to become nonunique or may change the characters which two classes have in common. The rest of this section is devoted to giving solutions to the above problems by presenting examples and a set of concepts which, when followed, eliminate conflicts.

The first two examples deal with the basic fa and with backup. These examples do not have to do with extensibility but deal with the basic structure of the fa as proposed by Gries [5].

Example 1:

Symbols to be parsed= {A | AB | ABC}

Character classes= {ϕ}



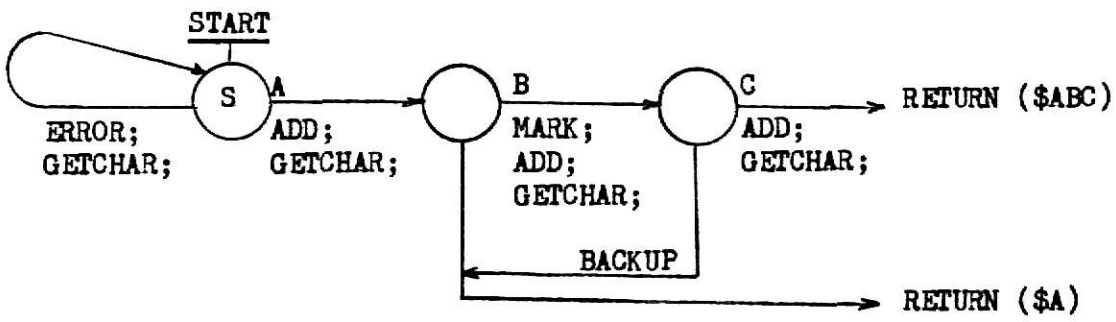
This example shows:

GENSCAN Concept 1-- Backup is not needed as long as the next source character allows determination of the uniqueness of the symbol.

Example 2:

Symbols to be parsed= {A | ABC}.

Character classes= {ø}.



This example illustrates:

GENSCAN Concept 2-- If the next source character cannot be used to determine uniqueness, then mark the last character in the current symbol and continue to try to parse the longest string. If parsing

the longest string fails, then remove all the characters from the current symbol back to the last mark and that is the symbol parsed.

DYNAMIC TRANSITIONS

In this section two examples are given which cover the configurations described in the previous section. With each example there are concepts given for the construction of the fa and for the changing of transitions after the fa is constructed.

The first example deals with the concepts that arise when characters or classes of characters are taken into consideration. The example covers the cases of both independent and unique characters.

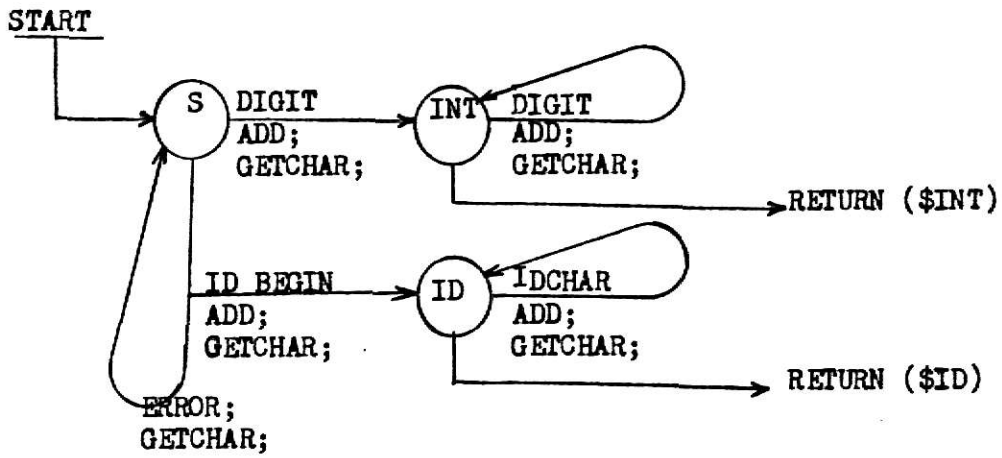
Example 3:

Symbols to be parsed=

{ID::= LETTER {LETTER | DIGIT},
INT::= DIGIT {DIGIT}}

Character classes=

{IDBEGIN::= LETTER,
IDCHAR::= LETTER DIGIT ,
DIGIT::= DIGIT}



IDCHAR is a independent class of characters since it does not matter which characters belong to this class. IDBEGIN and DIGIT are two unique classes since they cannot have characters in common. This example illustrates the following concepts:

GENSCAN Concept 3-- If a character class is independent of all other classes in the fa, then any character may belong to that class, and a character may be added to or deleted from an independent class without checking other classes in the fa.

GENSCAN Concept 4-- If a character class must be unique, then a character may be added to that class by using the following algorithm.

1. If the character is of any other unique character class, report a message indicating conflict and failure.
2. If the character is already a member of the class, report message indicating same.
3. If steps 1 and 2 fail, insert character into the proper class and report success.

A character may be removed from a unique class by simply removing it and reporting same.

The next example deals with symbols which are grouped by type, and each type has a class describing the characters which are allowed to start each class; but these two classes have some characters in common, therefore, an additional class is needed.

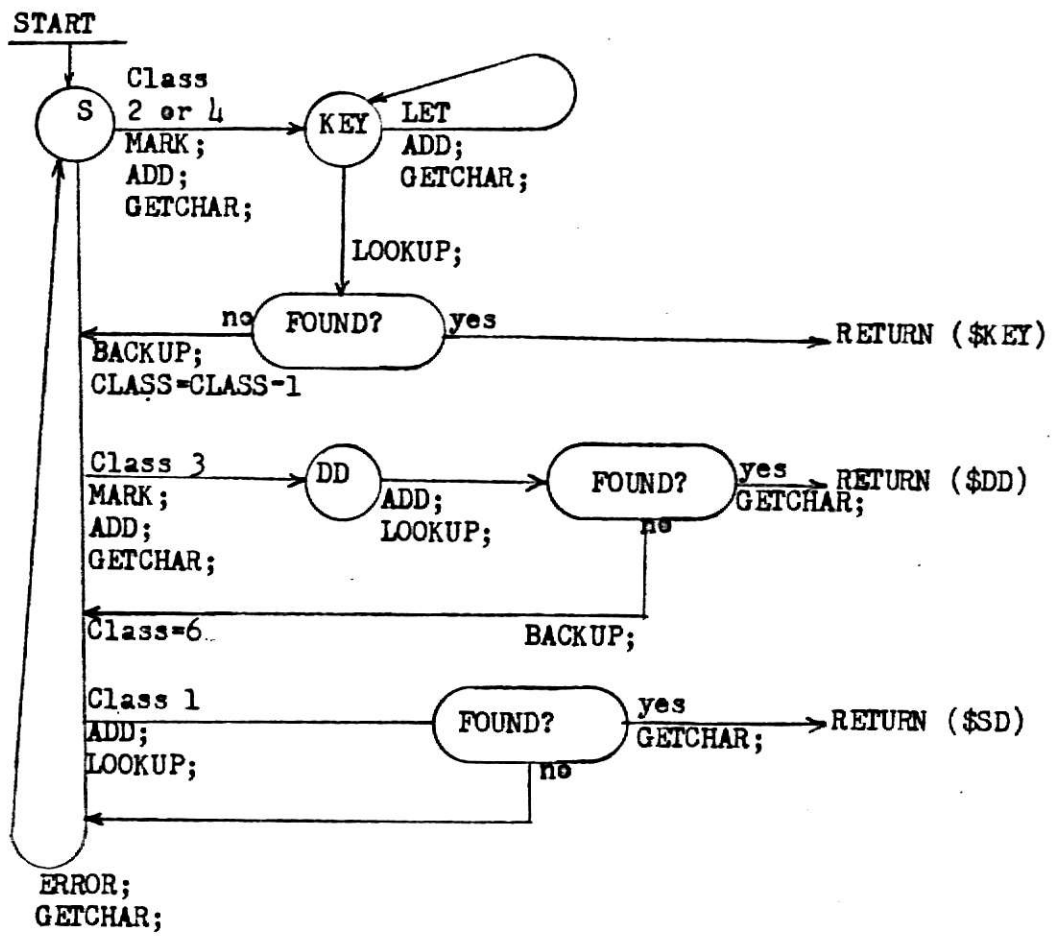
Example 4:

Symbols to be parsed=

{SINGLE DELIMITERS::= {* | + | - | / | . | = | ↑},
DOUBLE DELIMITERS::= {** | // | =: | || | >=} ,
QUOTED KEYWORDS::= {.ED. | /ABS/ | |NE| | 'LT' | "GT"}}

Character classes=

{CLASS 1= NOKEY-NODD::= {+ | - | ↑},
CLASS 2= YESKEY-NODD::= {. | ' | "},
CLASS 3= NOKEY-YESDD::= {* | = | >},
CLASS 4= YESKEY-YESDD::= {/ | '|'},
CLASS LET::= {A | B | ... | Z}}



This example illustrates several GENSCAN concepts. The first deals with lookup tables and when they are needed.

GENSCAN Concept 5-- If a group of symbols has a class grouping their starting characters and an internal code is needed for each symbol, then a lookup table is required to find the internal code.

The next concept deals with classes that have characters in common with other classes.

GENSCAN Concept 6-- If two or more character classes have characters which are common with one other class, add an additional class

to describe these common characters and build the machine so that it parses the largest or most common group first. As this string is being parsed, mark the first character and continue parsing until completion. If at any time parsing fails, backup and try parsing the next group.

At this point two additional problems arise. The first problem is that grouping start characters together into a class for a small group of symbols may be less efficient then parsing the individual strings when individual internal codes are needed. (Lookup tables are inefficient). The second problem is that grouping start characters together into a class which has characters in common with other classes may be less efficient, due to backup, then defining unique classes.

With the addition of tables comes the concept of allowing the designer or user the option of changing entries in these tables. It should be noted that the user has no direct control over the contents of the classes which describe the starting characters in this situation, but that the actual tables dictate what belongs in the classes.

GENSCAN Concept 7-- A symbol may be added to a table that requires uniqueness from some classes but not all classes by the following algorithm.

1. If the start character is not unique from those characters in classes requiring uniqueness, then report same and return failure.

2. If the start character conflicts with a character in a class describing another type of symbol, then remove that character from that class and place it in the common class. Add the symbol to the proper table and report success and return.
3. If the start character is not in the class describing the start characters for its type, then place the start character in that class, place the symbol in the proper table, report success and return.
4. Place the symbol in the proper table, report success and return. Note a user defined internal code may be placed in the table at the same time the symbol is.

GENSCAN Concept 8-- A symbol may be deleted from a table that has a class describing its start symbol with characters that are common to other classes by the following algorithm.

1. If the starting character is the same as any other starting character for an entry in that table, delete the symbol from the table, report success and return.
2. If step 1 is false, but the class describing the start characters for the table contains that start symbol, delete the character from the class, delete the symbol from the table, report success and return.
3. If steps 1 and 2 are false, then find the common class which has the start character in it and move that character to the class(es) that the symbol start character is common with, delete the symbol from the table, report success and return.

Two more functions are needed in order that the user may be informed as to the contents of the tables.

GENSCAN Concept 9-- Tables may be listed by reporting their contents.

GENSCAN Concept 10-- If the internal codes represented in a table are unique as far as range is concerned, then an avail function may be defined as follows:

1. Report the values of all internal codes not having entries in the table from the lowest to the highest value.

or alternately

1. Report the value of the first internal code not having an entry in the table.

The concepts described in this section give the designer the tools to construct a lexical analyzer which is dynamic in nature. The add and delete functions give the user the capabilities of inserting or changing the primitives of the language.

LANGUAGE SYNONYMS

When dealing with the subject of extensibility, one may ask "How can the language features be transmitted to non-professionals in different areas?" or "How can the vocabulary be redefined for different user communities?". One answer to these questions is synonyms. Synonyms allow the language to become flexible enough for many user communities. Synonyms are symbols which may be inserted in the language and which have the same meaning as delimiters or words already in the language.

The question arises of "Why handle synonyms in the lexical scanner and not in the symbol table?" The approach taken here is not that handling synonyms in the lexical processor is "the" way it should be done, but only that it is a way of approaching the problem and should be investigated. The GENSCAN approach is that since there are "lookup" tables in the scanner, then the addition of synonym tables for referencing is the most practical approach.

GENSCAN Concept 11-- When synonyms are needed and lookup tables are available in the scanner, the addition of synonym tables are of little cost to the overall system.

A sample lexical analyzer may have the following master lookup tables:

1. Keyword Table (KEYTAB) A table used to store all quoted and unquoted keywords along with their internal representations.
2. Single Delimiter Table (SDTAB) A table used to store all single delimiters (operators, grouping characters, and separators) along with their internal representations.
3. Double Delimiter Table (DDTAB) A table used to store all double delimiters (operators, grouping characters, and separators) along with their internal representations.

This same lexical analyzer then may have the following synonym tables:

1. Keyword Synonym Table (SYMKEY) A table of variable names which are used as synonyms for keywords, single delimiters or double delimiters and a pointer to the table entry for the symbol for which they are synonyms.

2. Single Delimiter Synonym Table (SYMSD) A table of single delimiters which are used as synonyms for keywords, single delimiters or double delimiters and a pointer to the table entry for the symbol for which they are synonyms.
3. Double Delimiter Synonym Table (SYMDD) A table of double delimiters which are used as synonyms for keywords, single delimiters or double delimiters and a pointer to the table entry for the symbol for which they are synonyms.

Example 5:

If a language had the following original definitions:

KEYWORDS= {GO, TO, .EQ.,...}

SINGLE DELIMITERS={+,=,...}

DOUBLE DELIMITERS={**, /*...}

and the following synonyms are added:

synonym	original
BRANCH	is a synonym for GO
PLUS	is a synonym for +
←	is a synonym for =
↑	is a synonym for **
=	is a synonym for .EQ.
<-	is a synonym for =
//	is a synonym for /*

The resulting table configurations after the synonyms are added are shown in Figure 2 .

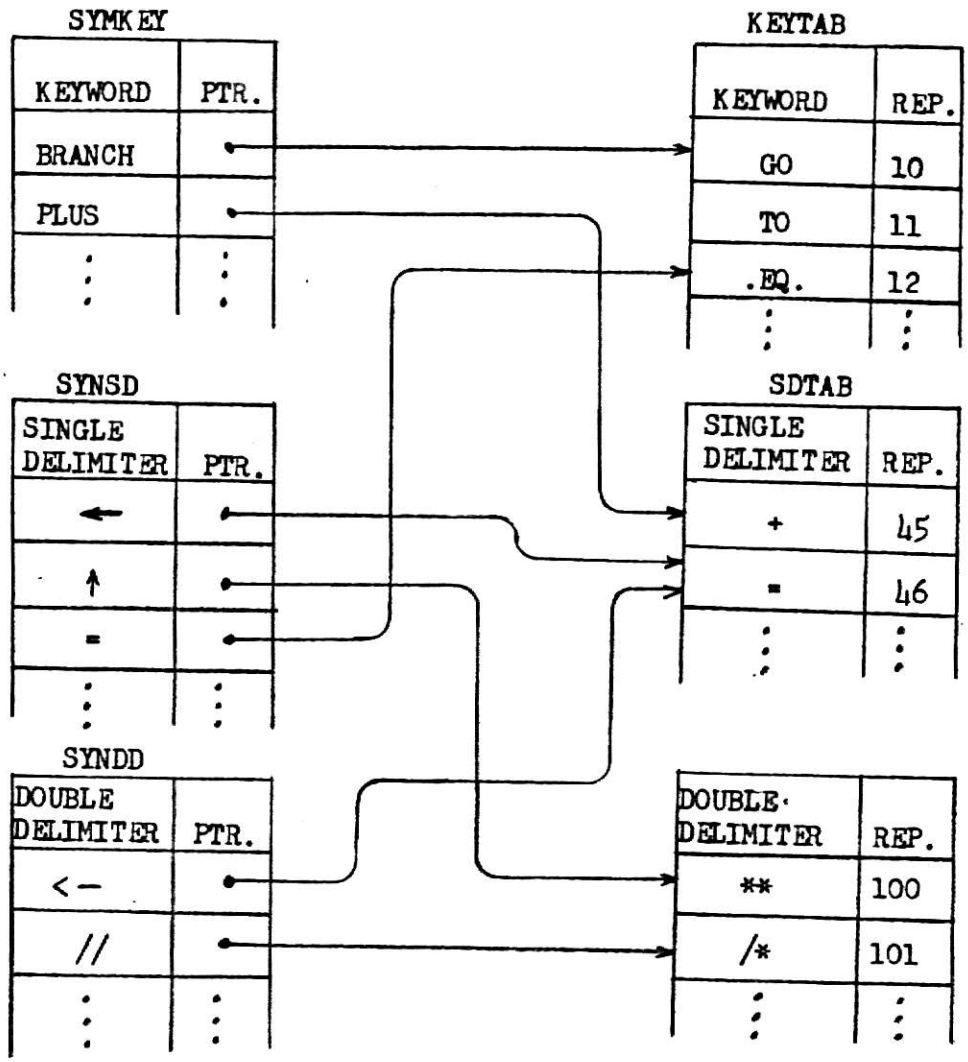


Figure 2 . Master and synonym tables.

It should be noted that the lexical processor should look up internal codes in the synonym tables before the master lookup tables, thereby insuring the proper meanings of synonyms.

HENCEFORTH FUNCTION

Now that the method of building synonym tables has been defined for adding synonyms to the language, a function is needed to build these tables.

GENSCAN Concept 12-- The HENCEFORTH function for building synonym tables is defined by the following algorithm.

1. Determine the type (keyword, double delimiter, single delimiter, etc.) of both the original symbol and the synonym symbol from their unique internal codes.
2. Add an entry to the proper synonym table (determined in step 1) for the symbol representing the synonym.
3. Point the pointer in the synonym table to the entry in the master lookup table representing the original symybol (determined in step 1).

Note that if the type of symbol has its own unique range of internal codes, step one can be accomplished in a much more efficient manner.

Now that the method of building the synonym tables has been defined, a formal definition of the lookup function is needed.

When synonyms are employed in the language, the following modified LOOKUP algorithm is needed.

1. Search the synonym table for the symbol.
2. If the symbol is found in the synonym table, then return the internal code for the entry pointed to by the synonym entry.
3. If the symbol is not found in the synonym table, then search the master table for the symbol.

4. If the symbol is found in the master table, then return its internal code.
5. If the symbol is not found in either table, return failure.

The henceforth function may be called by or may be part of the syntax driver. In either case it should be noted that the henceforth function is a dynamic function and must be interpreted when the henceforth statement is encountered.

MODULAR STRUCTURING

The next GENSCAN concept pertains to the structuring of the lexical scanner.

GENSCAN Concept 13-- A lexical scanner should be constructed of modules, where each module is a unit in itself and may be programmed as a macro which can be initiated from other programs.

This concept implies that if a library of all possible modules were constructed, then the designer of a lexical analyzer could build a scanner with a series of macro calls.

The structure illustrated in Figure 3 shows how one portion (the section dealing with numbers) of the lexical scanner may be modularly constructed.

The structure is a portion of a lexical scanner which is capable of parsing:

1. Integers of the form DIGIT {DIGIT}
2. Real numbers of the form ⟨INT⟩ . {DIGIT}

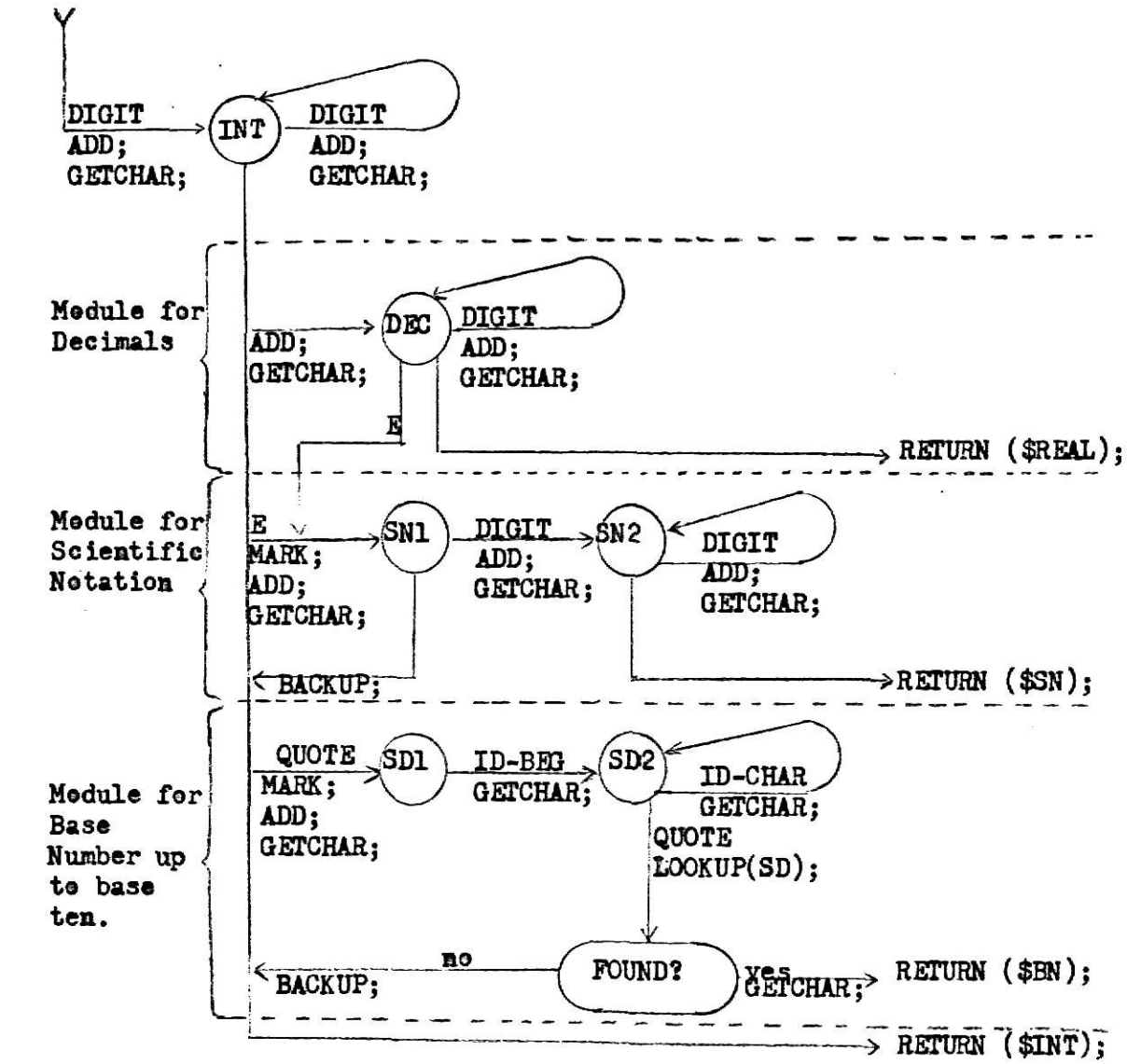


Figure 3 . Modular structuring of a lexical analyzer.

3. Scientific numbers of the form (⟨INT⟩ | ⟨REAL⟩) E ⟨INT⟩

4. Base numbers of the form ⟨INT⟩('BIN' | 'OCT')

This scanner is structured so that when an integer is parsed the next character determines if a call should be made. If that next character is a "." then, a call is made to the "real" routine (indicated by the branch labeled "." going into state DEC). The real routine then parses as far as it is able and then checks the next character to see if it is an "E". If the next character is an "E" then a call is made to the "scientific" routine (indicated by transition from state DEC to state SN1) If the next character is not an "E" then a "real" token is returned. If the "integer" routine finds the next character to be an "E" or a "QUOTE", then the proper routine is called. If while inside the "scientific" or "base number" routines, failure occurs, then the backup is performed and failure is reported to the "integer" portion (indicated by the backup branches). The "integer" portion then returns an integer token.

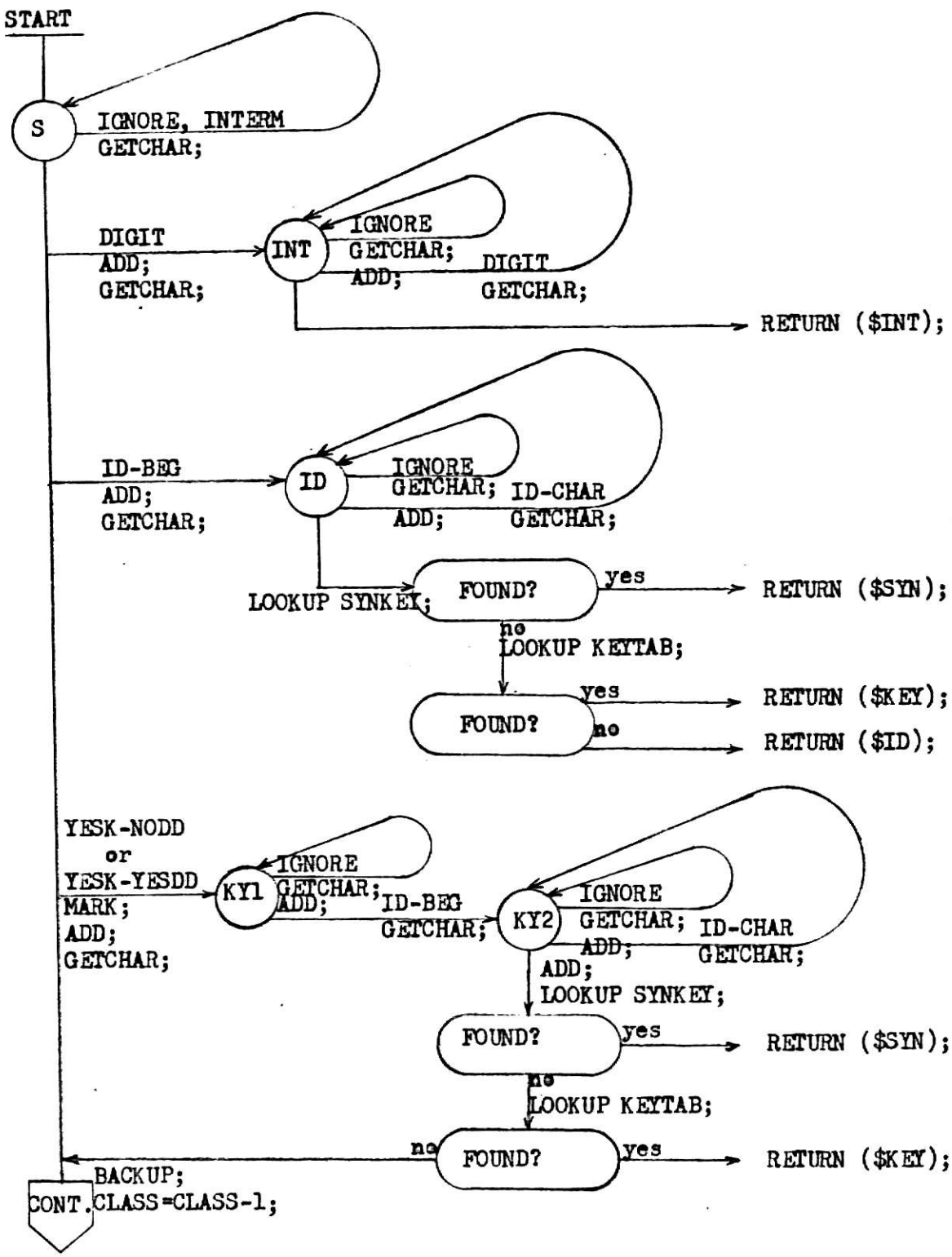
IV. TABLE CONSTRUCTOR

SAMPLE LEXICAL ANALYZER

The ability to dynamically change tables allows the user and the designer the ultimate in extensibility as far as lexical processors are concerned.

Since the functions used to dynamically construct tables and classes vary with the application, a sample scanner will be used to show these applications. Add, delete, and interrogate algorithms used to construct tables will be defined for the sample. The sample scanner uses the concepts described in the previous sections and has master and synonym tables like these described in the section on synonym tables. If a set of rules are set down for the contents of classes and tables, then the construction function may be defined. Table 2 lists the restrictions on classes and table entries for the sample language. In addition, the scanner has the class definitions shown in Table 1. The scanner is represented by the state diagram shown in Figure 4. The semantic routines have the same meanings as those described in the section on semantics.

It should be noted that, although this sample scanner does not parse many of the primitives used in most languages, it does have all of the basic fa configurations and illustrates all of the constructor functions needed to build a more complex constructor. The designer using the concepts described in this chapter should have no trouble constructing a lexical analyzer and table constructor capable of handling almost any language primitive.


START
S
IGNORE, INTERM
GETCHAR;
DIGIT
ADD;
GETCHAR;
INT
IGNORE
GETCHAR;
ADD;
DIGIT
GETCHAR;
RETURN ($INT);
ID-BEG
ADD;
GETCHAR;
ID
IGNORE
GETCHAR;
ADD;
ID-CHAR
GETCHAR;
LOOKUP SYNKEY;
FOUND?
yes
RETURN ($SYN);
no
LOOKUP KEYTAB;
FOUND?
yes
RETURN ($KEY);
no
RETURN ($ID);
YESK-NODD
or
YESK-YESDD
MARK;
ADD;
GETCHAR;
KY1
IGNORE
GETCHAR;
ADD;
ID-BEG
GETCHAR;
KY2
IGNORE
GETCHAR;
ADD;
ID-CHAR
GETCHAR;
ADD;
LOOKUP SYNKEY;
FOUND?
yes
RETURN ($SYN);
no
LOOKUP KEYTAB;
no
FOUND?
yes
RETURN ($KEY);
BACKUP;
CONT.
CLASS=CLASS-1;

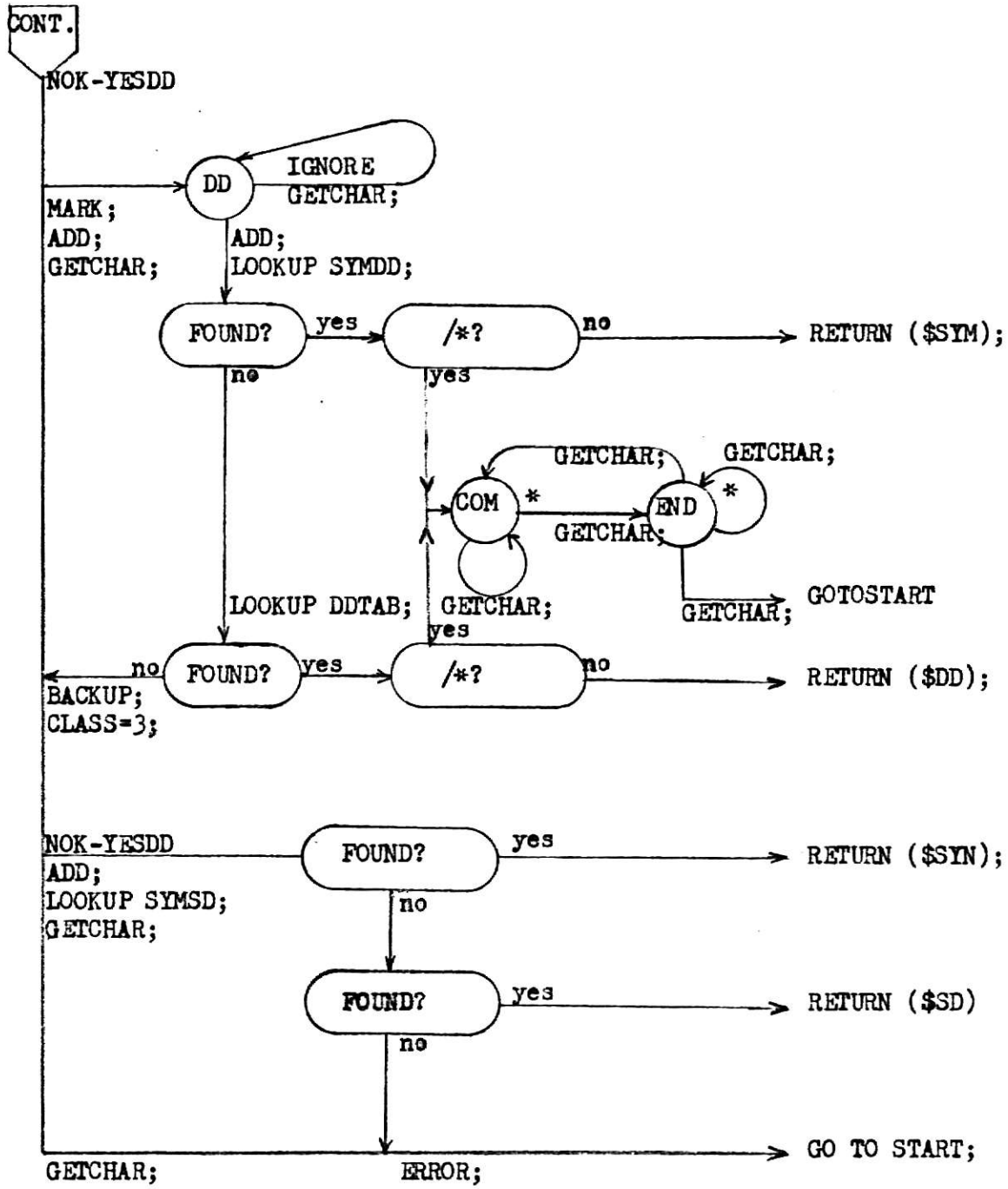


Figure 4. State diagram for sample scanning algorithm continued.

CLASS	DEFINITION	RESTRICTION	INITIAL VALUE
IGNORE	Characters ignored or deleted.	Unique from all other classes.	"carriage return"
INTERM	Character that ends symbols but otherwise ignored.	Unique from all other classes.	"blank"
DIGIT	Characters in "Integer" symbol.	Unique from all but ID-CHAR.	0,1,2,...,9
ID-BEG	First character of "Identifier" symbol.	Unique from all but ID-CHAR.	A,B,C,...,Z
ID-CHAR	2nd,3rd,...characters of "Identifier" symbol.	Independent of all classes except IGNORE and INTERM.	0,1,2,...,9,A,B,C,...Z
NOK-NODD	Single delimiter characters not beginning Quoted keywords or Double delimiters.	Characters unique to single delimiters.	+,-,↑
YESK-NODD	Single delimiter characters which begin Quoted Keywords but not a double delimiter.	Characters common to single delimiters and keywords.	. , ', "
NOK-YESDD	Single delimiter characters which begin a double delimiters but not keywords.	Characters common to single delimiiters and double delimiters.	*,=,>
YESK-YESDD	Characters which begin single delimiters, double delimiters and quoted keywords.	Characters common to single delimiters, double delimiters and quoted keywords.	/,|

Table 1. Class Definitions.

TABLE	USE	INITIAL ENTRIES	INIT. CODE RANGE
KEYTAB	Quoted and unquoted keywords.	DO, TO, .EQ., /ABS/, NE ,'LT', "GT"	10 to 44
SDTAB	Single delimiters.	*,+,-,/,.,=,↑	45 to 99
DDTAB	Double delimiters.	**,//,=:,|| ,>=	100 to 124
SYNKEY	Keyword Synonyms.	NO ENTRIES	NA
SYNSD	Single delimiter synonyms.	NO ENTRIES	NA
SYNDD	Double delimiter synonyms.	NO ENTRIES	NA

Table 2. Table Definitions.

CONSTRUCTOR ADD AND DELETE FUNCTIONS

The purpose of this section is to define the functions necessary to build a table constructor for the sample lexical analyzer described in the last section. Each of the functions will be discussed in detail, and flow charts representing each type of function are given.

Before the constructor functions can be described, some of the notations and functions used must be defined. The variables KEY, SD, and DD will represent the character strings for keywords, single delimiters and double delimiters, respectively. The variable represented by INT-NO contains the internal code number. The notation SD_1 represents the first character of the single delimiter in the variable SD and the notation $KEY_{2\text{-}LTH}$ represents the second character through the last character in the keyword represented in the variable KEY. The functions used throughout this section are explained next.

1. CONFLICT (character, class) is a function which checks if the character or characters passed in the first argument conflict with any of the characters in the class passed in the second argument and returns a true or false value.
2. REPORT-CONFLICT is a function which reports the situation of the last conflict.
3. REPORT-SUCCESS is a function which reports that add or deletion was accomplished.
4. INSERT (table, symbol, (internal code number)) is a function which performs one of the tasks described below.
   A. If the table passed as the first argument is a synonym table, then the symbol passed as the second argument is inserted in the given table with an internal code of the third argument and success is reported.
   B. If the table is a master table, then the symbol is inserted at the entry dictated by the internal code and success is reported. If that entry is not available, then the conflict is reported.
   C. If the argument representing the internal code is omitted, then the symbol is inserted at the next available entry and the internal code number is reported.
5. MOVE (character, class1, class2) is a function which moves the character passed as argument from the class passed as the second argument to the class passed as the third argument.

6. PLACE (character, class) is a function which places the character passed as the first argument in the class passed as the second argument.
7. REMOVE (character | symbol, class | table) is a function which removes the character or symbol passed as the first argument from the class or table passed as the second argument.
8. SEARCH (table | class, symbol | character) is a function which searches the table or class passed as the first argument for the symbol or character passed as the second argument and returns a true or false value.

The first function to be discussed is that which changes the entries in classes. Since changes in classes which represent the starting characters for quoted keywords and delimiters are dictated by entries in their tables, these classes will be discussed later. The class functions are designed using GENSCAN Concepts 3 and 4.

The IGNORE and INTERM classes are similar in that they must be unique from all other classes. Therefore, they are discussed together. The function to add a character to the IGNORE or INTERM is described as follows:

1. If the character is already in the given class, report success, and return.
2. Otherwise, if the character is in any other class, report the conflict, and return.
3. Otherwise, insert the character in the given class, report success, and return.

The DIGIT and IDBEG classes are similar therefore, they will be discussed together. The add function for DIGIT and IDBEG is described as follows:

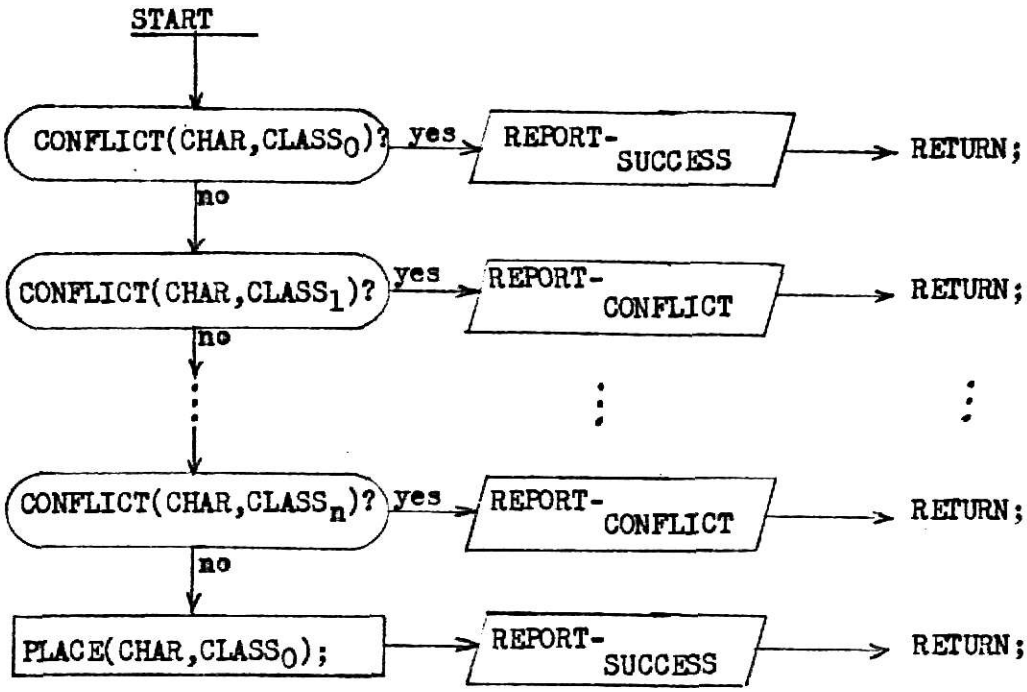
1. If the character is already in the given class, report success, and return.
2. Otherwise, if the character is in any other class except IDCHAR, report the conflict and return.
3. Otherwise, insert the character in the given class, report success, and return.

The add function for the class IDCHAR is described as follows:

1. If the character is already in the IDCHAR class, report success and return.
2. Otherwise, if the character is in the class IGNORE or the class INTERM, report the conflicts and return.
3. Otherwise insert the character in the IDCHAR class, report success, and return.

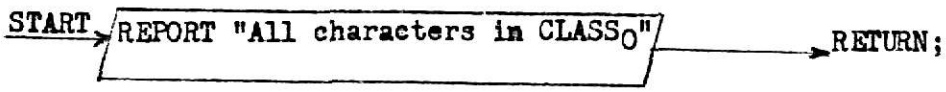
Since all of the add functions are similar, a generalized flow chart for the functions is given in Figure 5a. The classes from which the given class must be unique are represented by $CLASS_1$ through $CLASS_n$. The character to be inserted is in the variable called CHAR and the class in which it is to be inserted is called $CLASS_0$.

The delete functions and the interrogate functions are the same for all of the classes given above and are illustrated in Figure 5b and 5c respectively.



(a) Generalized ADD-CHAR-TO-CLASS function.

(b) Generalized DELETE-CHAR-FROM CLASS function.



(c) Generalized INTERROGATE-CLASS function.

Figure 5. Flow charts for generalized class constructor functions.

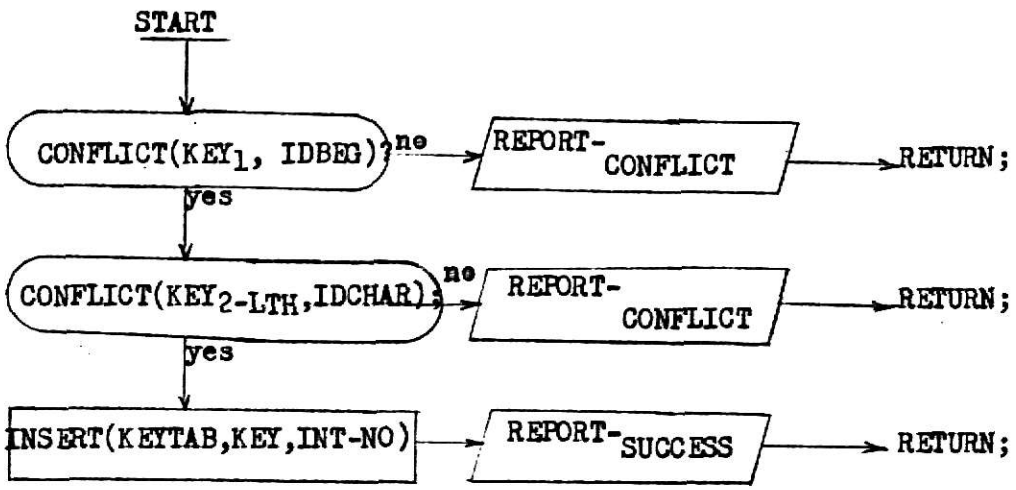
The add functions for keywords may take on one of two forms depending on the designer's needs. First, there may be separate functions for keywords and quoted keywords and second, the keywords may dictate the contents of classes or the classes may determine legal keywords.

Two add functions for keywords will be discussed here. The first add-a-keyword function will be for unquoted keywords which must conform to the characters in the IDBEG and IDCHAR class. In this case the character classes dictate the legal characters allowed in the keyword. This add function uses GENSCAN Concepts 5 and 7. This add function is described as follows:

1. If the keyword does not begin with a character in the IDBEG class, report conflict, and return.
2. Otherwise, if all but the first character of the keyword are not in the IDCHAR class, report conflict, and return.
3. Otherwise, if proper space is not available in KEYTAB, report same, and return.
4. Otherwise, insert in KEYTAB the keyword with the proper internal code representation and return.

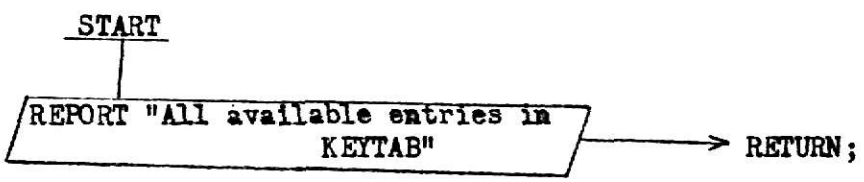
The flow charts for this function, a delete function and an avail function are given in Figure 6.

The second add-a-keyword function will be for quoted keywords only. These quoted keywords consist of a keyword as described in the first part which is quoted with a legal single delimiter. In this example the quotes dictate what is contained in some of the classes and the



(a) Simple ADD-KEYWORD function.

(b) Simple DELETE-KEYWORD function.



(c) Simple INTERROGATE-KEYTAB function.

Figure 6. Flow charts for simple keyword constructor functions.

contents of the body of the keyword is dictated by the contents of the IDBEG and the IDCHAR classes. This add function uses GENSCAN Concepts 3 through 7. This add-a-keyword function is described as follows:

1. If the keyword does not begin and end with a proper single delimiter, then report conflict, and return.
2. Otherwise, if the keyword's body is not a proper keyword, then report conflict, and return.
3. Otherwise, if the keyword begins with a character in the YESK-NODD or the YESK-YESDD class, then enter the keyword in KEYTAB, report success, and return.
4. Otherwise, if the keyword begins with a character in the NOK-YESDD class, then move that character from the NOK-YESDD class to the YESK-YESDD class, report success, and return.
5. Otherwise, move the character from the NOK-NODD class to the YESK-NODD class, report success, and return.

The flow chart illustrating this function is given in Figure 7. The delete function for this type of keywords is described as follows:

1. If the keyword is not listed in KEYTAB, report same, and return.
2. Otherwise, if the first character of the keyword is the same as the first character of any other keyword, then remove keyword from KEYTAB, report success, and return.
3. Otherwise, if the first character of the keyword is the same as any character that starts a double delimiter, then move the character

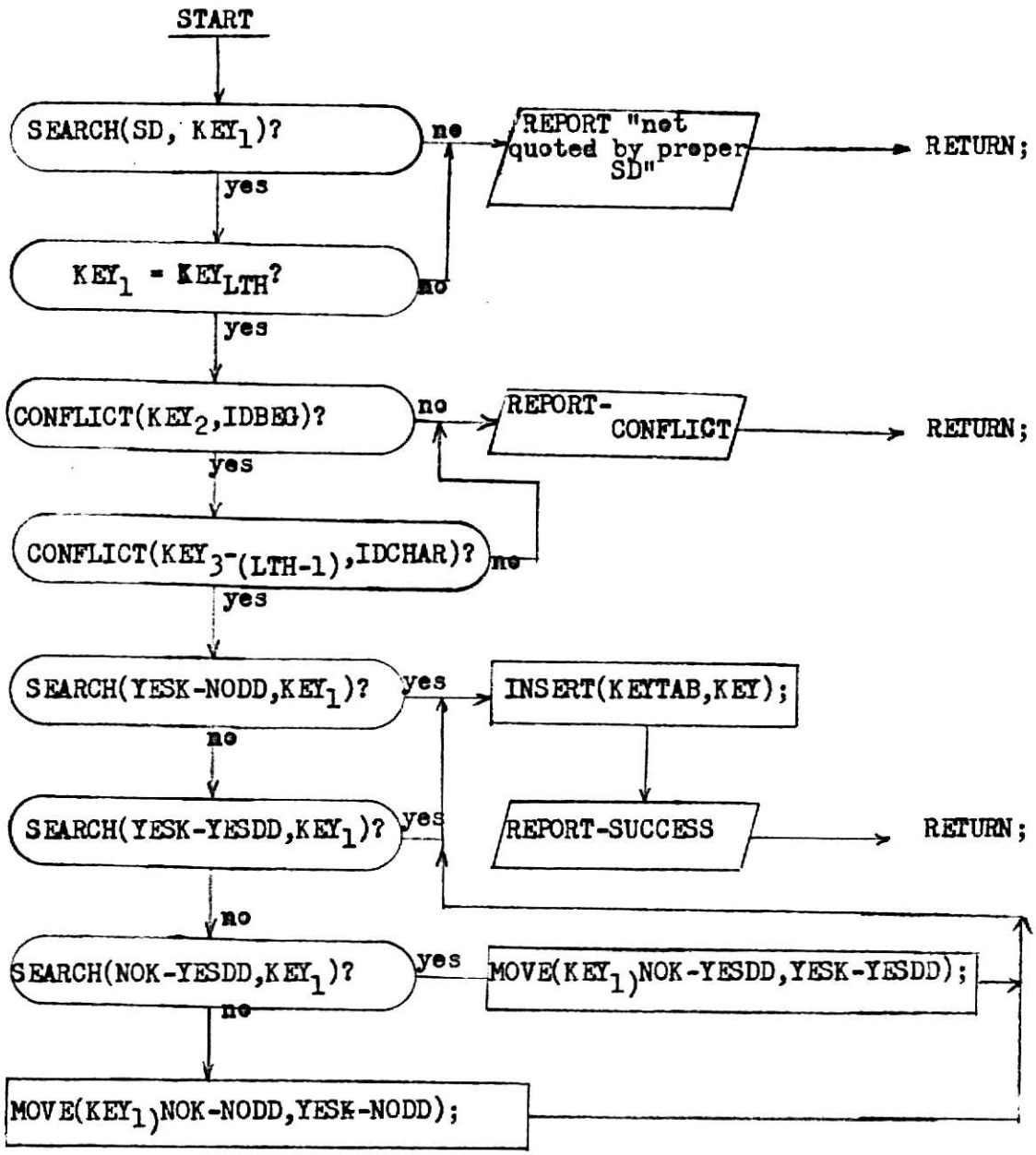


Figure 7. Complex ADD-A-KEYWORD function.

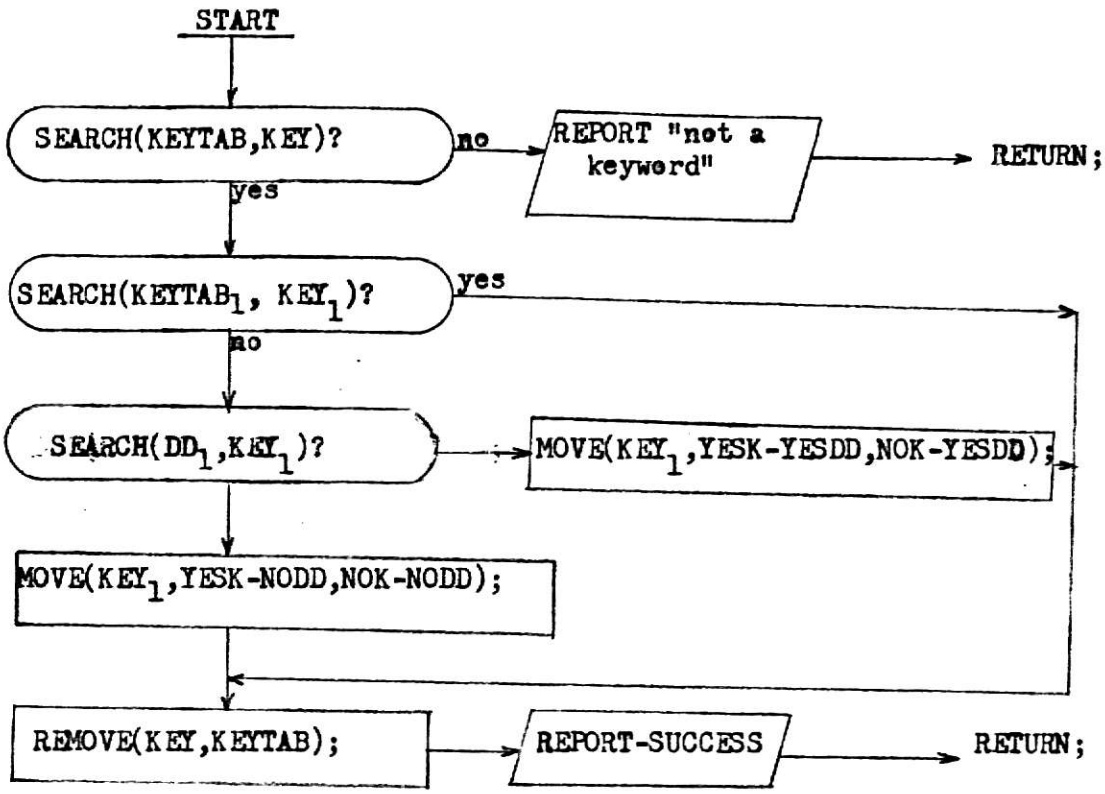


Figure 8. Complex flow chart for DELETE-KEYWORD function.

from the class YESK-YESDD to the class NOK-YESDD, remove the keyword from KEYTAB, report success, and return.

4. Otherwise, move the first character of the keyword from the class YESK-NODD to the class NOK-NODD, remove the keyword from KEYTAB, report success and return.

The flow chart for this delete function is given in Figure 8. The avail function for this type of add-a-keyword is the same as the one for simple keywords and the flow chart is illustrated in Figure 6c.

The functions for adding and deleting single delimiters use the GENSCAN Concepts 3 - 8. The flow chart for the add-a-single-delimiter is illustrated in Figure 9a. The add-a-single-delimiter function is

described as follows:

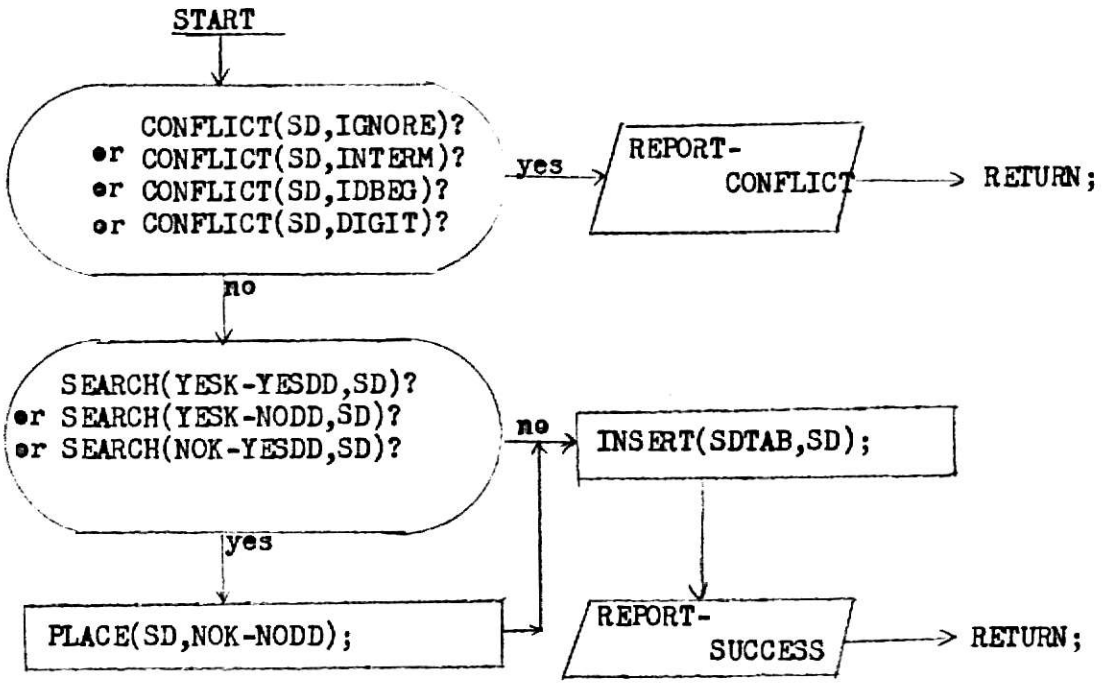
1. If the single delimiter conflicts with a character from the IGNORE, INTERM, IDBEG, or DIGIT classes, report same and return.
2. Otherwise, if the single delimiter is in one of the YESK-YESDD, YESK-NODD, or NOK-YESDD classes, then insert the single delimiter in SDTAB, report success, and return.
3. Otherwise, insert the single delimiter in the class NOKEY-NODD, insert the single delimiter in SDTAB, report success, and return.

The flow chart for deleting a single delimiter is illustrated in Figure 9b and the function is described as follows:

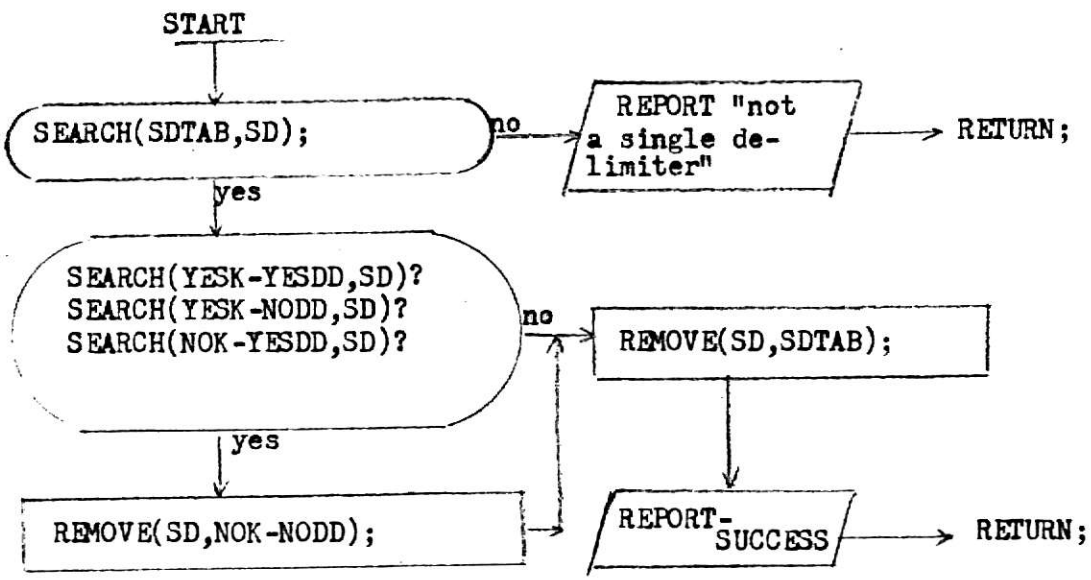
1. If the single delimiter is not in SDTAB report same and return.
2. Otherwise, if the single delimiter in one of the YESK-YESDD, YESK-NODD, or NOK-YESDD classes, then remove the single delimiter from SDTAB report success, and return.
3. Otherwise, remove the single delimiter from the class NOKEY-NODD, and the table SDTAB, report success and return.

It should be noted that the single delimiter function as described above allows a single delimiter to be removed when it may be used to quote a keyword. This has an affect on the previously defined quoted keywords.

The functions for adding and deleting double delimiters use GENSCAN Concepts 3 - 8. If the rule is that a double delimiter must be made up of two single delimiters, the add-a-double-delimiter may be defined as follows:



(a) ADD-SD.



(b) DELETE-SD.

Figure 9. Flow charts for adding and deleting single delimiters.

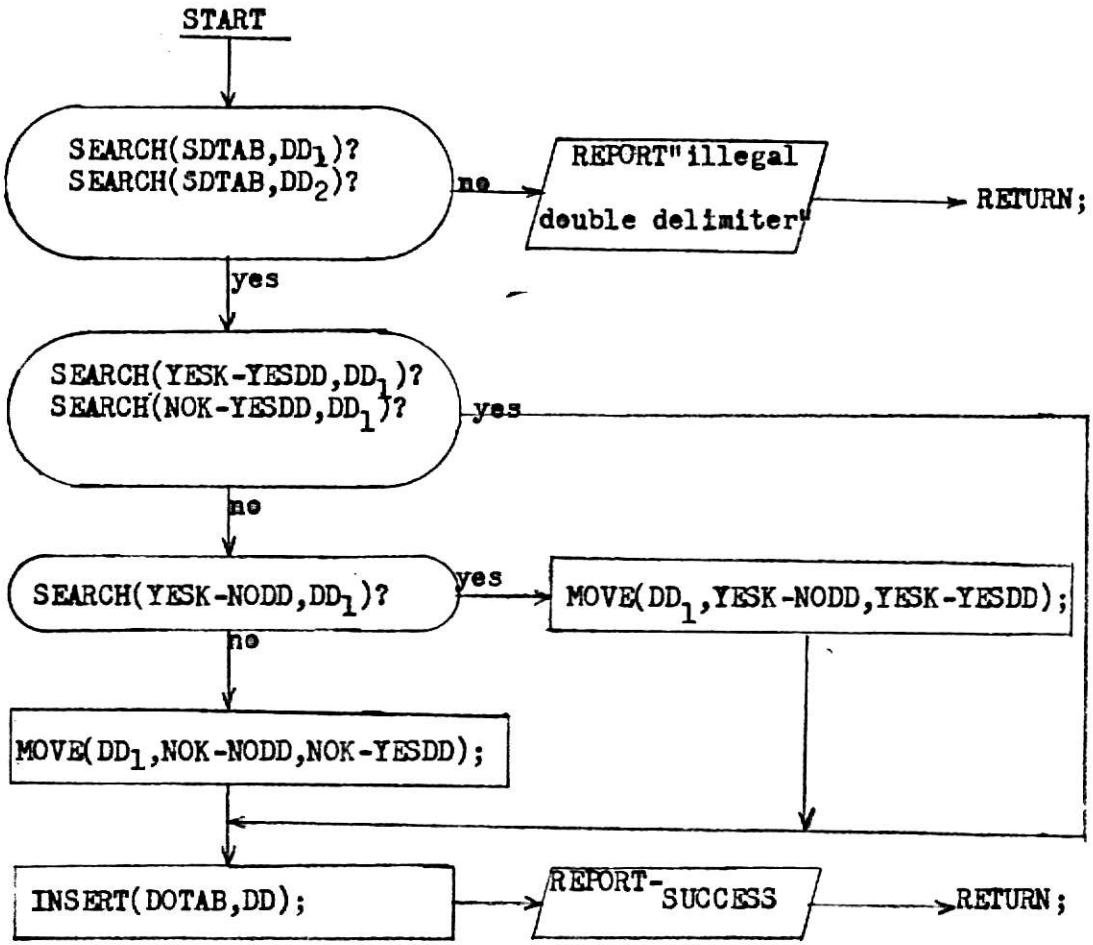


Figure 10. Flow chart for ADD-DD function.

1. If both characters in the double delimiter are not single delimiters, then report same, and return.
2. Otherwise, if the first character of the double delimiter is in the class YESK-YESDD or the class NOK-YESDD, then insert the double delimiter in DDTAB, report success, and return.
3. Otherwise, if the first character of the double delimiter is in the class YESK-NODD, then move the character from the class YESK-NODD to the class YESK-YESDD, insert the double delimi-

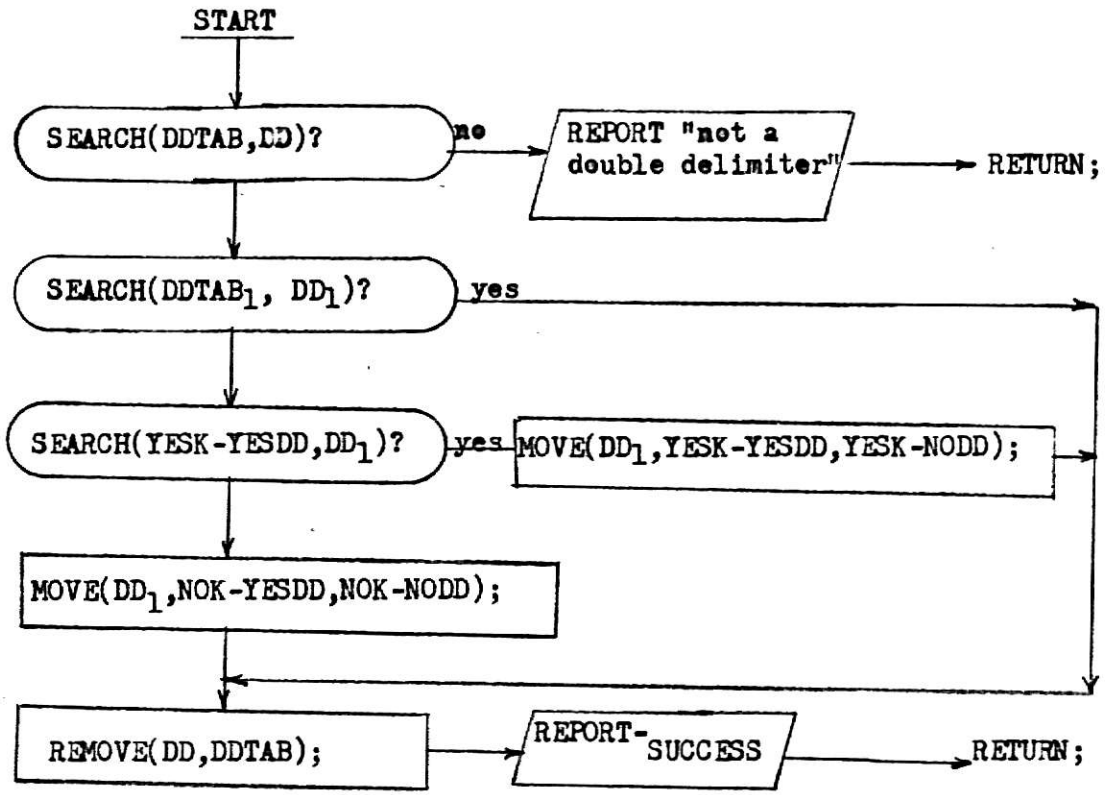


Figure 11. Flow chart for DELETE-DD function.

ter in DDTAB, report success and return.

4. Otherwise, move the first character of the double delimiter from the class NOK-NODD to the class NOK-YESDD, insert the double delimiter in DDTAB report success and return. The flow chart for the delete-a-single-delimiter function is illustrated in Figure 10. The delete-a-double-delimiter flow chart is given in Figure 11 and the function is defined as follows:

1. If the double delimiter is not in DDTAB, report same and return.
2. Otherwise, if the first character of the double delimiter is the same as another first character in any other double delimiter,

remove the double delimiter from DDTAB, report success, and return.

3. Otherwise, if the first character of the double delimiter is in the class YESK-YESDD, then move the character from the class YESK-YESDD to the class YESK-NODD, remove the double delimitter from DDTAB, report success, and return.
4. Otherwise, move the first character of the double delimiter from the class NOK-YESDD to the class NOK-NODD, remove the double delimiter from DDTAB, report success, and return.

The interrogate functions are approximately the same as the previous interrogate functions and will not be discussed in this paper.

The constructor functions described in this section may be used by the language designer with direct calls to the functions or they may be used by the language user through interpreter routines. For a overall view of the structure of an extensible lexical analyzer and a description of how the table constructor routines fit in and are used, see the diagram illustrated in Figure 12. It should be noted that in order to use the function described in this section, the user must know the names of the scanner's classes and tables. It is expected that the language designer would know these names, but this is too much to ask of the language user. It is suggested that the interpreter routines which use the add, delete, and interrogate function help resolve the names of the classes and tables for the user.

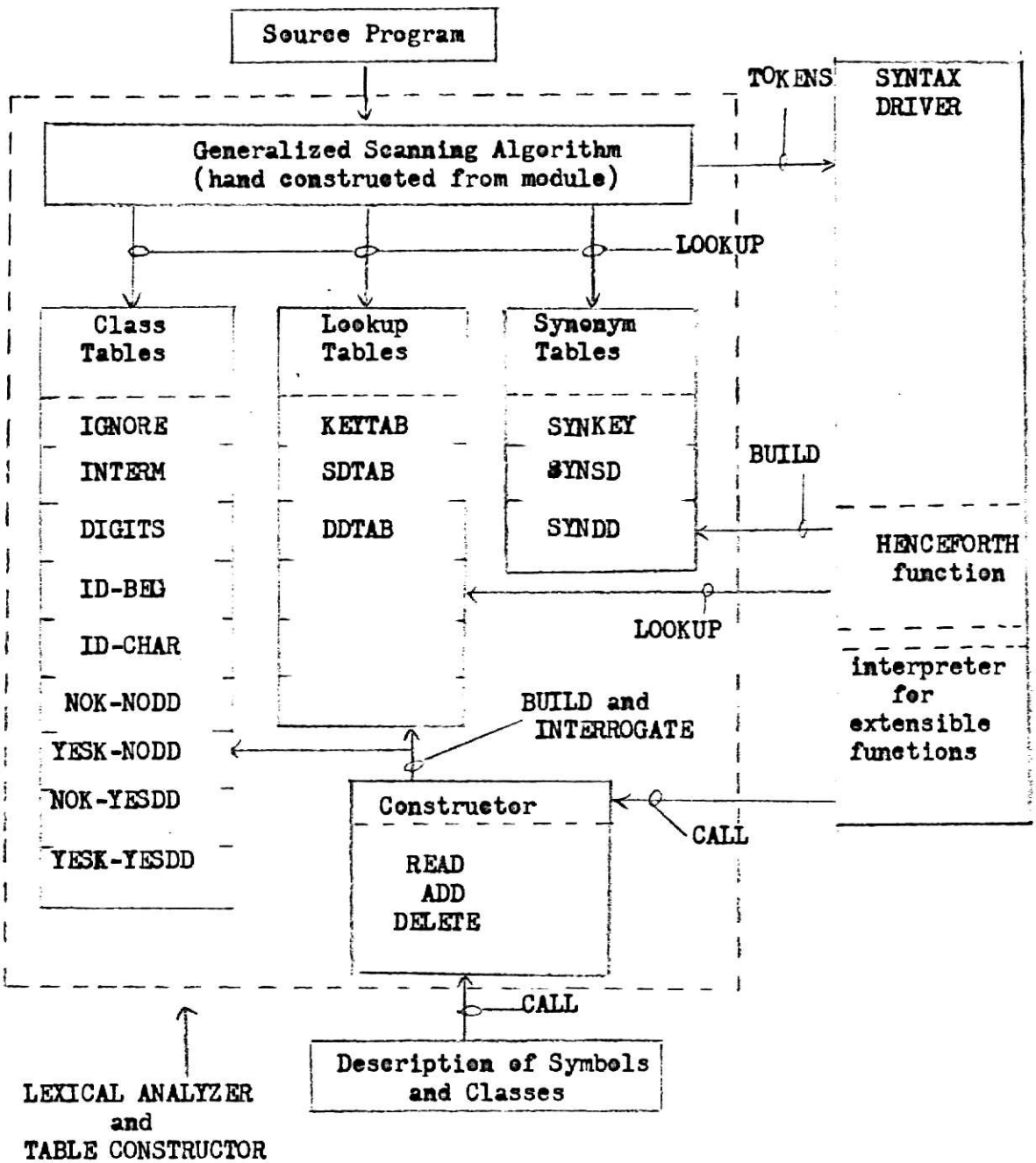


Figure 12. Diagram of Extensible Lexical Analyzer.

SYNONYM CONSTRUCTOR

The HENCEFORTH function allows the language user the option of giving a synonym for any primitive used in the language. The HENCEFORTH function uses the variables SYN, SYN-NO, and OLD-NO to represent the synonym, the internal code number for the synonym, and the internal code for the original or old symbol respectively. These internal codes are determined when the HENCEFORTH statement is scanned by the lexical analyzer. Since the function described in this section requires that the type of the synonym be known, the types of symbols must have unique ranges of internal codes. The HENCEFORTH function described here requires the definition of a function called SYNTAB. SYNTAB is a function which determines the type of symbol from the internal code passed as its argument. SYNTAB returns the name of the synonym table into which this type of symbol can be inserted. The HENCEFORTH function assumes that space is available for a new entry in the synonym table. The HENCEFORTH function is described as follows:

1. Determine the synonym table in which the synonym belongs.
2. Insert the synonym in the proper synonym table along with the old internal code, report success, and return.

The flow chart for this function is illustrated in Figure 13. The way the HENCEFORTH function fits into the structure of the extensible lexical analyzer is illustrated in Figure 12.

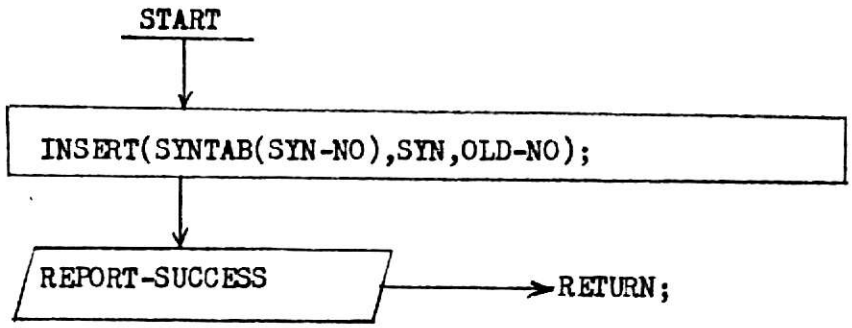


Figure 13: Flow chart for HENCEFORTH function.

V. CONCLUSION

GENSCAN VS. RWORD

If a lexical analyzer designed with GENSCAN Concept is coded using a transition table, then the resulting fa would be approximately the same as a scanner built by AED-RWORD. The basic differences between the two machines are as follows:

1. A GENSCAN scanner is a routine which is called by the rest of the translator, while a RWORD scanner is the main procedure which calls the rest of the translator.
2. Each time the GENSCAN scanner is called the procedure is entered at the start state, but each time the RWORD scanner is envoked the start point is the next state after the last state which sent a symbol to the rest of the translator.
3. A GENSCAN scanner always parses the longest possible symbol (unless a special routine is written), while a RWORD scanner may not parse the longest symbol (what is parsed may rely on the last symbol parsed).

If the designer has a good library of GENSCAN modules available, then the building of a scanner from these modules would require the designer to do more program logic then is required with RWORD.

EFFICIENCY

An efficient hand-coded scanner is approximately twice as fast as the same scanner built by RWORD. The RWORD scanner itself does not use back up but the checking for back up is passed on to the rest

of the translator. Since RWORD scanners themselves don't use back up they would be more efficient than a GENSCAN scanner, but the efficiency of the total system would not vary that much. The efficiency lost due to the proposed extensibility features are directly proportioned to the lookup algorithm used in searching tables. If the synonym feature were added to any translator some type of lookup is inevidable.

ALTERNATIVES

There seems to be two alternatives to allowing the proposed extensibility to be handled by the lexical analyzer. The first alternative is to redesign the translator whenever the language is extended which is absurd and the second alternative is to handle extensions and synonyms through the symbol table. Since at this point in time a system using GENSCAN concepts has not been implemented, the two approaches cannot be discussed as far as efficiency is concerned.

FUTURE DIRECTION

The next future step, other then the implementation of a GENSCAN lexical analyzer, is a system which allows the user to change the structure of the fa used as a lexical analyzer. With a proper library of the proposed modules it is conceivable that interpreter routines could be written to allow this dynamic structuring of the lexical analyzer.

APPENDICES

APPENDIX A

AUTOMATA THEORY

FINITE-STATE MACHINES

The general approach used in RWORD and GENSCAN is based on the theory of finite-state automata or sequential machines. The RWORD system requires the user to specify the lexical properties of his language in a limited subset of regular expressions. These regular expressions are then translated into a deterministic finite-state automaton. Most of the diagrams used in this paper to describe GENSCAN are graphical representations (state diagrams) of finite-state machines. In order to show proof of validity of the RWORD and GENSCAN systems, the relationships between regular expressions and finite-state machines must be defined. Therefore, an understanding of regular grammars and their associated parsing machines (finite-state automata) is essential to the understanding of this paper. The purpose of the appendix on automata is not to present a complete examination of automata theory with rigorous proofs; but, to give a general overview of finite-state machines which is essential to the understanding of this paper.

In the following definitions of regular grammars, regular expressions, state diagrams and finite-state machines are given. The main concern of this paper is to understand how RWORD and GENSCAN operate. Therefore, some results are not formally proven. (For formal proofs, see Kain [7]).

REGULAR GRAMMAR

Throughout this paper the need arises for a basic language to use in examples. The language, called G, is made up of symbols found in most programming languages. These symbols fall into one of the following types:

identifiers (including keywords)

reserved words (which are a subset of the identifiers)

integers

single character delimiters (+, -, (,), /, etc.)

double character delimiters (//, /*, **, :=, etc.)

These symbols can be described by the following simple rules:

```
⟨identifier⟩          ::= letter | ⟨identifier⟩ letter
                           | ⟨identifier⟩ digit
⟨integer⟩             ::= digit | ⟨integer⟩ digit
⟨single delimiter⟩    ::= + | / | *
⟨double delimiter⟩    ::= ⟨SLASH⟩ / | ⟨SLASH⟩ *
                          | ⟨AST⟩ *
⟨SLASH⟩               ::= /
⟨AST⟩                 ::= *
```

Of course the rules could be simpler, but the rules are written so that each rule has the form

U ::= T or U ::= VT

where T is a terminal and V is a nonterminal. A grammar with such rules is called a type 3 or regular grammar. The syntax of most programming

language symbols can be specified in such a form, so that it would be advantageous to find an efficient way of parsing sentences of a regular grammar.

STATE DIAGRAM

Consider the regular grammar specified by the identifier production of G,

⟨identifier⟩ ::= letter | ⟨identifier⟩letter | ⟨identifier⟩ digit

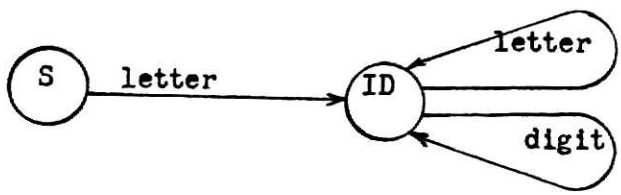


Figure 14. State diagram of identifier recognizer.

The state diagram to recognize the identifier production is shown in Figure 14. The circles or nodes are the states and the arrows are the transitions. Each state represents a unique condition-of-tension in reading the character string, and all such legal possible conditions are represented. At each state the machine reads a new character and changes to the new state indicated by the transition. If the reading operation comes up with any character other then these indicated on the transitions, an error condition is indicated. More formally, in a state diagram, each nonterminal V in the regular grammar is represented by a node or state, and it has a start state S. For each production U ::= T

in the grammar, there is a directed arc labeled T from the start state S to a state Q. For each rule U ::= VT in the grammar there is an arc labeled T from state V to state U.

The use of the state diagram to parse or recognize a string x as an identifier is as follows:

1. Begin in the start state S (it is the initial "current" state). Starting with the leftmost character of x, iterate step 2 until the right end of x is reached.
2. Scan the next character of x. Proceed along an arc labeled with that character emanating from the current state to the next current state.

If at any iteration of step 2 there is no such arc, x is not a identifier and the machine stops. If we reach the end of x, then x is a identifier if and only if the last current state is ID.

DETERMINISTIC FINITE-STATE MACHINES

In order to be able to manipulate state diagrams, the need arises to formalize the concepts of: the states, the input characters, the start state S, a mapping M which, given the current state U and input character T, tells what the next state is, and the final state Z.

The definition of a (deterministic) finite-state automaton (fa) is a 5-tuple (K, VT, M, S, Z) where:

1. K is an alphabet of elements, called states;
2. VT is an alphabet called the input alphabet (the characters which can appear in a string or sentence);

3. M is a mapping (or function) from K x VT into K (if M (Q, T) = R, then when in state Q with incoming character T, switch to state R).
4. S (which must be in K) is the start state; and
5. Z is a nonempty set of final states, all of which must be in K.

fa are machines having only a finite number of internal states that can be used for memory and computation. At any point in the reading of the input character string this machine will be in some unique state. Upon reading the next input character the machine will change to a new state as dictated by the particular character read. A more formal definition of running the fa (or state diagram) with an input string x is accomplished by extending the mapping to:

M(U,E) = U for all states U;

M(U,Tt) = M(M(U,T),t) for each t in VT* and T in VT.

The first line means that if the input character is the empty symbol, the machine stays in the same state. The second line indicates that when in state U, with input string Tt, we apply the mapping M to get to state P=M(U,T) with input string t and then apply the mapping M(P,t). A string is accepted by an fa if M(s,t)= P where P is a final state.

TRANSITION GRAPHS

An fa with states S1,...,Sn and input characters T1,...,Tm can be represented by an n by m matrix B. The element B(i,j) contains the letter k of the state Sk such that M(Si,Tj) = Sk. Adopting the convention that S1 is the start state, and can have a list of the final states

in a vector. Such a matrix is sometimes called a transition matrix, since it indicates how to switch from one state to another. A complete description of a machine to recognize an identifier described above is shown in Figure 15.

	input string			
state	letter	digit	other	output
S	ID	HALT	HALT	Fail
ID	ID	ID	HALT	Accept

Figure 15. Transition graph for identifier.

NONDETERMINISTIC FINITE-STATE MACHINES

Trouble is encountered when constructing an fa if G contains two rules

S ::= T R ::= T

V ::= UT and W ::= UT

with the same right parts. For then there will be two arcs labeled T emanating from U in the state diagram, and the mapping M will not be single-valued. An fa constructed from such a diagram is called a nondeterministic fa and is defined as:

A nondeterministic fa, or nfa, is a 5-tuple (K,VT,M,S,Z) where

1. K is an alphabet of states;
2. VT is the input alphabet;
3. M is a mapping of K x VT into subsets of K;
4. S in K is the set of start states; and
5. Z is the set of final states (which are all in K).

Again, the important difference here is that the mapping M yields a set

of states instead of a single state. The second difference is that several states may be start states. We extend the mapping M to K x VT* as before by defining

$$M(U,E) = \{U\}$$

and

M(Q,Tt) is the union of the sets M(P,t) for P in M(Q,T) for each T in VT and t in VT*.

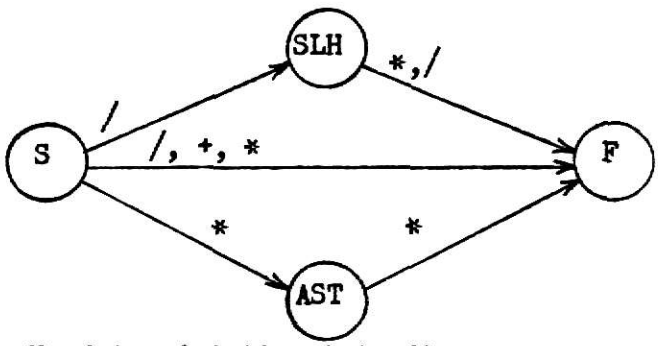


Figure 16. Nondeterministic state diagram.

The state diagram for a nfa to recognize the productions for a single delimiter and a double delimiter in grammar G is shown in Figure 16.

where S is the start state and F is the final state. The problem here is that at the start state S there is more then one arc labeled with a "/", so it is impossible to tell which path to take.

A technique employed to develop a deterministic machine from a nondeterministic machine is: Whenever a state transfers to more than one other state on a given character, represent this indecision by creating a single compound state having the combined properties of the states

transferred to. When this process is continued systematically to completion, the resulting machine will contain states which transfer to a unique state on each character. A more formal approach is the let F = (K,VT,M,S,Z) be an nfa accepting a set of strings L. Define the fa F'= (K',VT,M',S',Z') as follows:

1. The alphabet of states consists of all the subsets of K. We denote an element of K' by S1,S2,...,Sl where S1,S2,...,Si are states of K.
2. The set of input characters VT are the same for F and F'.
3. We define the mapping M'by

$$M'(S1,S2,\ldots,Si,T) = R1,R2,\ldots,Rj$$

where

$$M(S1,S2,\ldots,Si,T) = R1,R2,\ldots,Rj$$

4. Let S S1,...,Sn . Then S' S1,...,Sn .
5. Let Z be Sk,Sj,...,Sl . Then Z'= Sk,Sj,...Sl .

Then the set of strings accepted by F' is the same as that for F. The state diagram for an fa to recognize the production for a single delimiter and a double delimiter in the grammer G is illustrated in Figure 17.

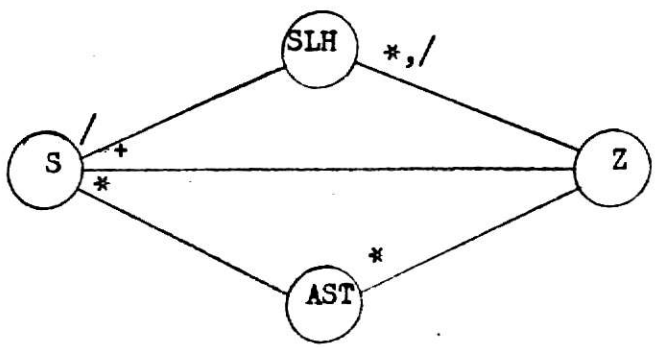


Figure 17. Deterministic state diagram. Where S is the start state and SLH AST and Z are the final states.

REGULAR EXPRESSIONS

Notations used through out this paper will deviate from standard Backus Normal Form (BNF) in the following ways:

1. Braces { and } are metasymbols used to bracket a string and mean zero or more occurences of that string. If the closing braces are followed by a number, this indicates the maximum number of occurences of that string. The BNF for the integer production is ⟨integer⟩ ::= digit | ⟨integer⟩digit. The same production is written ⟨integer⟩ ::= digit{digit} using the metasymbols. If the maximum length of the integer is defined to be five then the production is written as integer ::= digit{digit}4.
2. The metasymbol | inside braces is used to indicate a choice. Using the metasymbols to rewrite the BNF representation of the identifier production, ⟨identifier⟩ ::= letter|⟨identifier⟩letter|⟨identifier⟩digit, produces ⟨identifier⟩ ::= letter {letter | digit}.
3. Brackets are used to indicate an optional string and are equivalent to using braces followed directly by a 1.
4. In the right part of production rules, the concatenation operator has precedence over alternation. When necessary, parentheses are used to override this normal precedence.
5. When a language is used which contains a terminal which is also a metasymbol, then the terminal is distinquished by using quote marks around it.

There is a close connection between right parts of production using the above described metasymbols and regular expressions. With the exception of the symbol ϕ (the empty set) the meaning of a regular expression is similiar to that for the set of right parts of rules for a nonterminal. For example, the production ⟨integer⟩ ::= digit digit generates a digit followed by any number of digits, while the regular expression digit {digit} defines the same set of strings. From the previous discussion, it is obvious that regular expressions are just another method of describing a regular grammar.

A review of what has been stated in the preceding sections on finite-state machines leads us to the conclusion that an fa can be constructed from a description of the grammar given in regular expressions, and in fact this is what RWORD does.

LEXICAL ANALYZERS AND FINITE-STATE MACHINES

A lexical analyzer is a processor which recognizes the simple syntactic strucures of a language. If these simple structures can be represented by a regular grammar, then they may be recognized by a deterministic finite-state machine. The output from a lexical analyzer is the terminal symbols or primitives for the remaining syntactic structure which is recognized by the syntactic analyzer. Obviously, a machine which does nothing but read characters and change states is not of much value. However, these machines provide the structure upon which the actual processors are built. To make an fa function as a processor, various

operations must be carried out each time the machine changes states. These operations consist of the semantics for the scanner. A scanner may be programmed as a separate pass which performs a complete lexical analysis of the source program and which gives to the syntax analyzer a table containing the source program in an internal symbol form. Alternatively, it can be a subroutine called by the syntax analyzer whenever the syntax analyzer needs a new symbol. This produces the relationship of lexical and syntax analysis displayed in Figure 18.

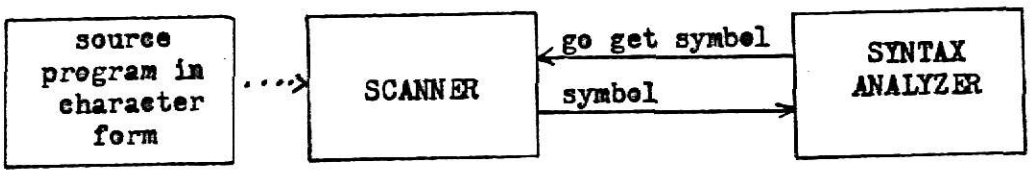


Figure 18. Basic relationship of scanner to rest of translator.

CONFLICTS

The difference between symbols and higher constructs is vague at times. For example, consider both integers and real numbers of the form

⟨integer⟩ . ⟨integer⟩

as symbols, or consider an integer to be a symbol and a real number to be a high-level construct. To this point a symbol was considered to be the token which can be recognized by a simple recognizing mechanism which, upon reading each character, the processor can determine unambiguously if that character starts a new item or if it should be included as part of the same item as the previous character. If a lexical analyzer is con-

fined to only this type of operation, its usefulness is extremely restricted. For some languages it cannot be determined whether a character belongs to the current symbol or begins a new symbol until several more characters have been read. FORTRAN IV illustrates this problem. Consider the following character strings:

Strings	Item Structures	Number of Items
2.EQ.K	2 .EQ. K	3
2.E10	2.E10	1

A simple processor does not allow these two strings since it is necessary to read past the "E" before the disposition of the "." can be determined. A complex lexical analyzer must cope with this type of look-ahead.

A scanner builds an internal representation of each symbol. In most cases this is a fixed-length integer (a byte, halfword, word, etc.). The rest of the compiler can process these integers much more efficiently than the variab e length strings which are the actual symbols. However, the symbol itself may be needed by the semantic analyzer, so it must be stored somewhere. The solution is to output two values; the first is the internal representation, and the second is the actual symbol itself or a pointer to it. (It is sometimes easier if the scanner keeps a table of all different symbols and returns a fixed integer as the second value (the index of the entry in the table)). Throughout the rest of this paper it is assumed that the lexical analyzer passes two values to the rest of the translator. Following this convention allows us to have one less concept included in examples, thereby making their structure simpler and easier to understand.

LEXICAL SEMANTICS

An example of the state diagram for the lexical analyzer of G will help illustrate the semantic insertions made to the finite-state machine. Assuming that G has three reserved words (BEGIN, ABS, and END), and comments of the form /* COMMENT */ are allowed, where Table3 gives the internal representations of the symbols.

INTERNAL REPR.	SYMBOL	MNEMONIC NAME	INTERNAL REPR.	SYMBOL	MNEMONIC NAME
0	unidefined	$UND	5	END	$END
1	identifier	$ID	6	/	$SLH
2	integer	$INT	7	+	$PLS
3	BEGIN	$BGN	8	*	$AST
4	ABS	$ABS	9	//	$SLSL

Table 3. Internal representation of symbols.

In the state diagram the mnemonic names will be used in place of the internal representations to make the diagram easier to read. There are several points to be discussed here. The label DIGIT is an abbreviation for the labels "0,1,2,...,9". That is, DIGIT represents the class of digits. This is done to keep the diagram simple. Similarly, the label LETTER represents the class of letters "A,B,...,Z" and DELIM represents the class of single character delimiters "+" and "*". Note that "/" is not in the class since it can occur in conjunction with "*" or "/" and must be handled in a special manner. Several of the arcs have no labels on them. These are the arcs to be taken if the character scanned is one that does not explicitly appear on an arc. For ex-

ample, when in state INT, as long as we scan a digit we stay in state INT. If a non-digit is scanned we proceed along the arc to RETURN. This example is a deterministic rather than a nondeterministic diagram. Thus, the problem of determining whether "/" begins a $SLH, $SLSL or a comment is solved by always going to the common state SLA.

The scanner will need the following variables and routines:

1. GETCHAR is a function which retrieves the next character of the source text and places it in CHAR.
2. ADD is a function which concatenates the current character to the string of characters in the buffer, called DATA, which is part of the token.
3. LOOKUP is a function which searches the table indicated for the data portion of the token and places the internal representation found in the I-REP portion of the token.
4. INIT is a function which makes sure that the present input character in CHAR is not a blank.
5. ERROR is a function which writes an error message that an undefined character has been encountered.
6. CHAR is a variable which always holds the character being scanned.
7. TOKEN is the variable returned by the scanner. It consists of two parts; the data portion and the internal representation.

These variables and functions are used to add semantics to the state diagram shown in figure 19 .

It should be noted that the lexical scanners described thus far in this appendix always returns the longest symbol possible. For example, if an

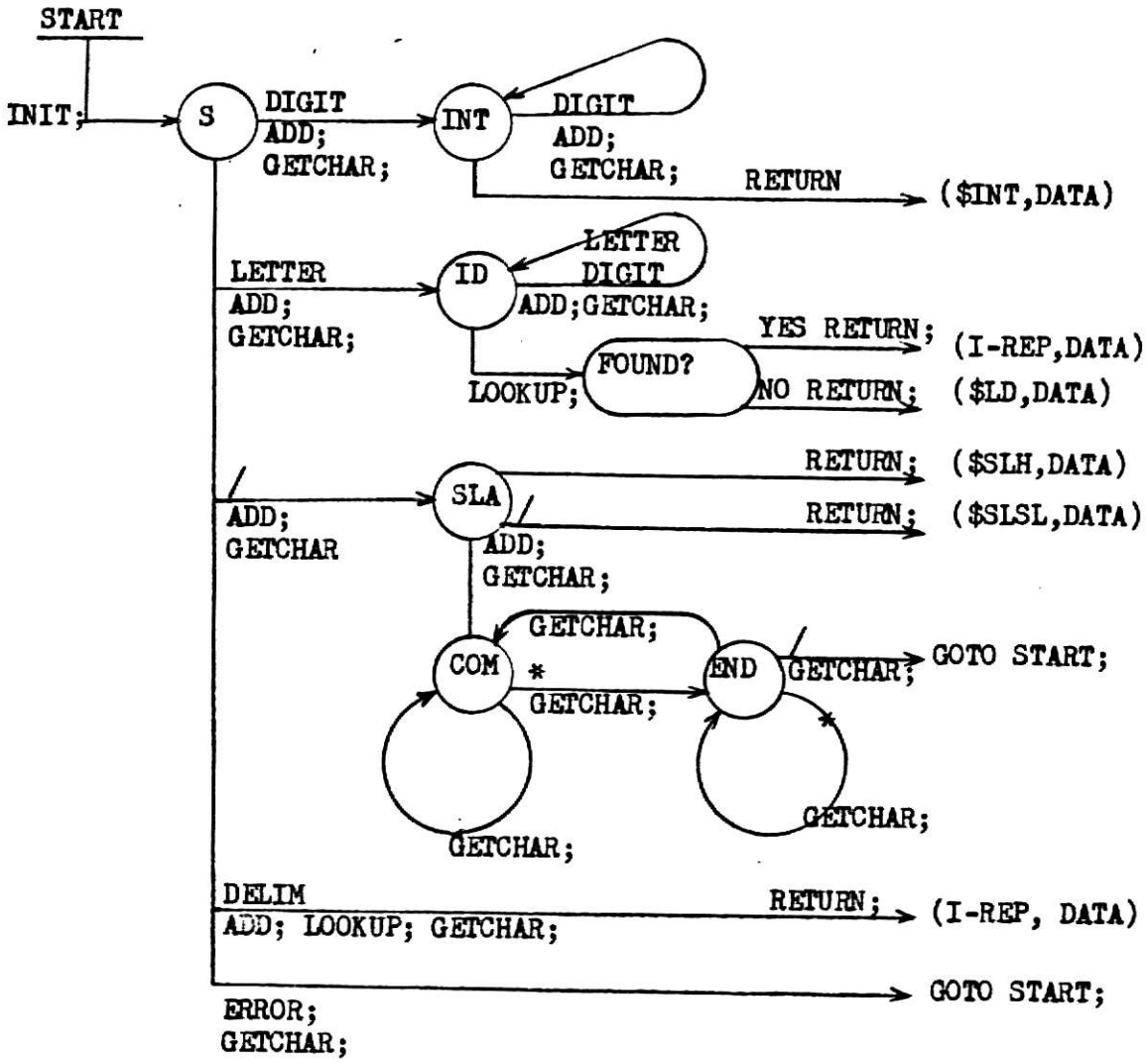


Figure 19. State diagram with semantics.

input string of A2 is encountered, the scanner will return the identifier A2 and not an identifier followed by an integer. This works well in most cases, but in the FORTRAN IV statement DO10I=1,20, the DO10I is not an identifier. Obviously, the scanner described thus far fails in some instances, but a small program segment to look ahead may be inserted to make a decision in such cases.

APPENDIX B

AED-RWORD LEXICAL PROCESSOR

AED SYSTEM

AED RWORD (Read-a-Word) is part of a software generating system called AED (Automated Engineering Design). AED was originally intended to design engineering problem-solving systems, but it is now used as a method of rapidly fabricating software packages. AED works on the theory that software can be tailored from the assembly of modular software components. The AED system is composed of the structure described in Figure 20.

The lexical processor is a specialized finite-state machine program which uses descriptions of how sequences of characters group together to form symbols. The parsing processor uses metalinguistic definitions to do both syntactic and semantic parsing of the problem. The modeling processor uses a description of the physical structure and the assembly language of the target machine to produce a symbolic machine-language program. Finally the analysis phase produces an assembler which is used to solve the problem. AED-RWORD composes a small portion of a completed system and has features which make it more generalized than are needed in a lexical scanner.

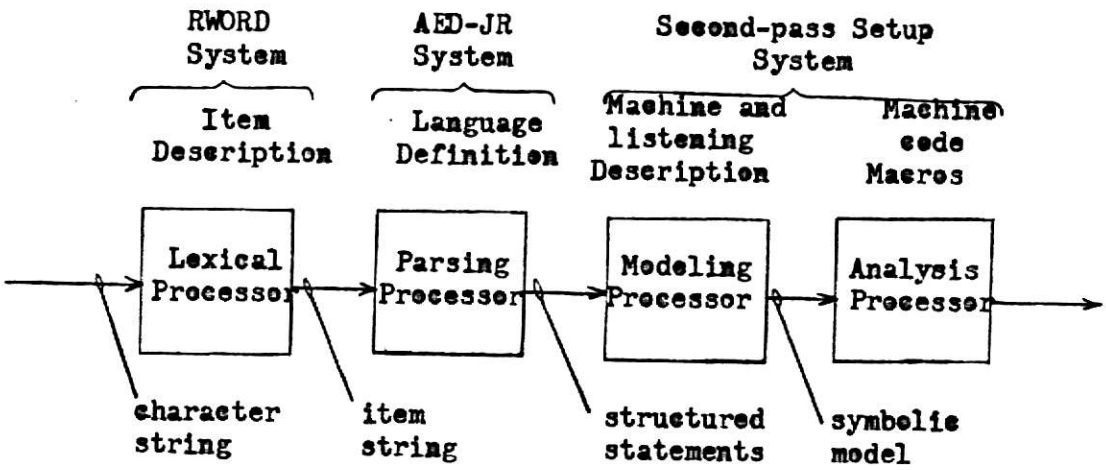


Figure 20. AED system structure.

AED-RWORD

The reason for including RWORD in this paper is that RWORD is "the standard" in lexical analyzers and presents a system to compare the proposed (GENSCAN) lexical analyzer against.

The general approach used in RWORD is based upon the theory of finite-state mahcines. The specialized lexical processor which is constructed by the RWORD System has the following characteristics:

> "At any instant, the machine will be in some unique state and some unique character will just have been read from the continuous string of input characters. This causes the machine to change its state to some other unique state, performing some unique action. The machine then is ready to accept the next character in a similar fashion." Ross 8.

It should be noted at this point that if the unique action is the same as the semantic action of the previous model discussed, then the AED lexical processor and the previous model operate in almost the same manner. The one exception being that once the AED scanner recognizes a symbol, it may not return to the start state.

The RWORD System accepts as input a description of the source symbols and a description of classes of characters. From these descriptions, the RWORD processor builds the machine described above. The RWORD System has the general structure illustrated in Figure 21.

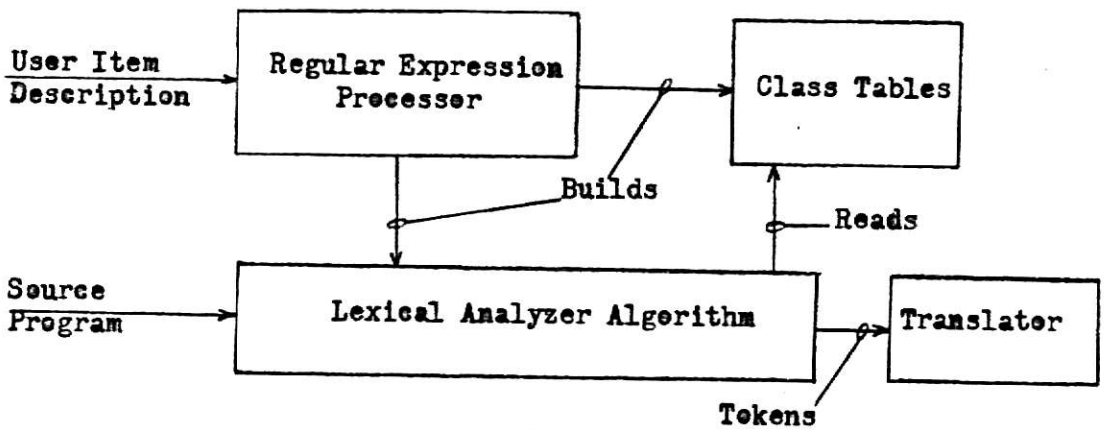


Figure 21. Basic RWORD structure.

RWORD INPUT

The input to the RWORD constructor is a program composed of two ALGOL-like blocks. The first block contains statements which describe classes of characters used in the source language. The block starts with a BEGIN statement and terminates with an END statement. The se-

cond block contains statements which describe the symbols used in the source language. The block starts with a BEGIN statement and terminates with an END FINI statement.

Class statements have the following format:

class-name= delimiter list-of-characters delimiter,

where

class-name is a mnemonic name for the class of characters and it will be used in regular expressions whenever that class of characters is needed;

delimiter is a delimiter where the first nonblank character after the equals sign is treated as the delimiter;

list-of-characters is a list of source characters which make up the class.

The RWORD constructor uses these statements to build tables which are consulted whenever the class is encountered in the lexical analyzer.

The symbol description statement is used by the RWORD constructor to determine how individual source characters and/or character classes are combined to form symbols. The format for the statement by which the user describes a symbol to RWORD is as follows:

item-name (code)= regular-expression $,

where

item name is a mnemonic name for the lexical type of symbol being described;

code is an integer which provides the internal representation for the lexical type;

$ is a terminator which indicates the end of the regular expression and of the symbol description statement;

regular-expression is e pression which describes the composition rules for symbols of a lexical type.

The regular expression operation used by RWORD has the same meaning as those described in the section on regular expressions. The difference here is that RWORD uses a different set of characters to represent these operations. Also, RWORD has the added operators of " ", "$", "'" which are used not as regular expression operators but as metasymbols in the regular expression statement.

The operators which are applied to individual characters or character classes are described in Table 4 .

Operator	Meanings	Example	Interpretation
/	Concatenate	A/B	A concatenated with B, i.e. AB
U	union, or	A U B	A or B
*	none or more of the preceding	B/A*	B concatenated with none or more A's, i.e. BAA or B or BAAA etc.
*n	none or more, but not exceeding n	B/A*2	B or BA or BAA
( )	the usual phrase separation	A/(B U C)	A concatenated with B or C, i.e. AB or AC
=	is defined as	EXAMPLE (1)= A U B $,	The item type named "EXAMPLE" with internal code "1" is defined as A or B
$,	item-description terminator		
'	the following character is not to be treated as an operator	'/ /A	/ followed by A
NULL	no character	A/(B U NULL)	AB or A [6].

Table 4. RWORD metasymbols.

RWORD OUTPUT

The output of RWORD is a table of information which controls the operation of the lexical processing procedures. The combination of the table and the procedures make up the lexical analyzer which will itemize a source character string in the manner specified by the description input. The lexical analyzer is activated by a call to it.

The analyzer returns one symbol for each call, by setting a pointer to a block of storage which contains the internal representation code, the string representing the symbol and the symbol's length.

Whenever a context-dependent situation arises, it is handled by constructing a separate processor for each context situation and switching between these processors when the appropriate character is encountered.

The RWORD system allows the user to specify procedures which are called by the processor. These procedures may be used to handle context situations, as described above, or they may be used to insert nonlexical functions where needed.

RWORD SYSTEM

The RWORD System is divided into three parts as illustrated in Figure 22. The first two parts are used to automatically construct tables used by the third part. In the first part, the regular expressions used to describe the source symbols are accepted as input. This first part generates parallel nondeterministic finite-state machines for each regular expression, builds a deterministic machine from the nondeterministic ones and finally generates an ordered sequence of calls on a library of standard programs (second part) which are used to build tables of machine code. These tables plus a library of standard lexical processing routines make up the third part of the system. This third part is the lexical analyzer which operates on the user's character string.

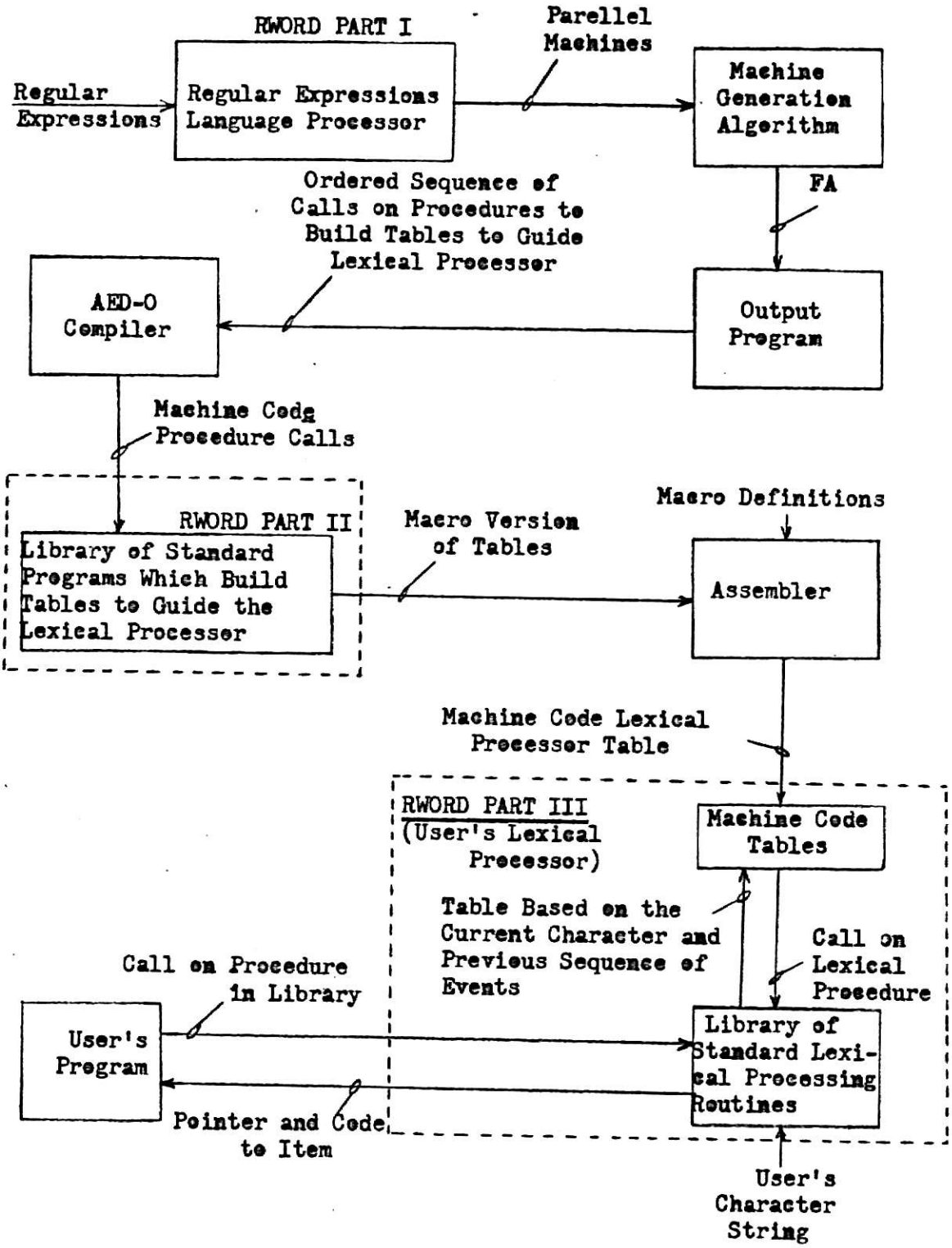


Figure 22. Detailed structure of the RWORD system.

RWORD PROGRAM

In order to understand the RWORD System, an example using a subset of G is in order. Only integers, identifiers, comments, and the delimiter "/" are used since they show all the concepts and any larger subset ould only tend to confuse the issue. Figure 23 shows the input program to RWORD and the resulting nondeterministic machines.

Program

```
BEGIN
LET=/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
DIG=/1234567890/
SPACE=/ /
SL=A/A
END
BEGIN
ID(1)=LET/(LET U DIG)*5$
INT(2)=DIG/DIG*$
SLH(6)=SL$
REMARK=(10,COMM) SL/'*$
IGNORE=SPACE$
END FINI
```

Resulting Nondeterministic Machine

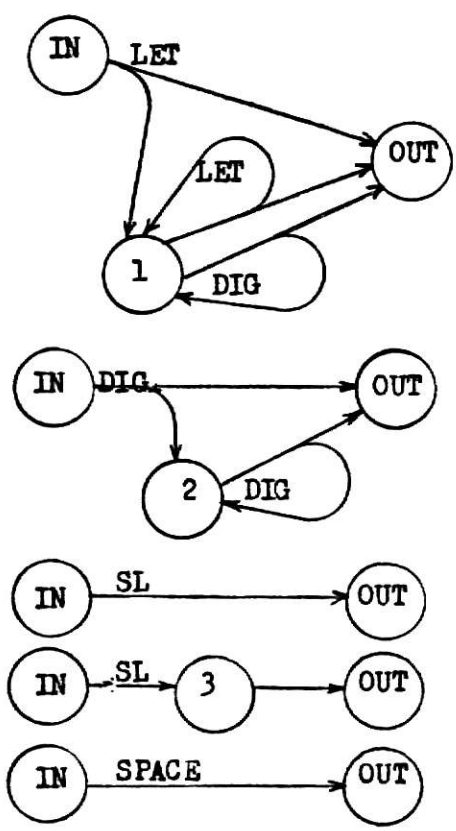


Figure 23. RWORD program and resulting machines.

The regular expression language processor reads in the program shown in Figure 23 and constructs non-deterministic machines for each regular expression given in the input program. These machines are represented as separate machines in the diagram, but actually they are one large machine since they have the states IN and OUT in common. Using these nondeterministic machines, the Machine Generation algorithm develops a deterministic machine. The technique used by the Machine Generator is that whenever a state transfers to more than one other state on a given character, represent this indecision by creating a single compound state having the combined properties of the states transferred to. In the example, the "IN" state for a slash and a comment will go to a combined state which will be an output state if the next character is not an "*".

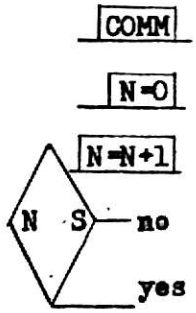
The State Diagram for the deterministic machine with the semantics added is shown in Figure 24.

The building of macro tables to represent the deterministic machine is outside the scope of this paper; therefore, there will be no further discussion of this part of the RWORD constructor.

RWORD uses the same semantics described previously; but the actions are represented on a state diagram in a different manner. The conventions used to describe the semantics in RWORD state diagrams are:

IN — Put the character in the character buffer

5 — Report all the characters in the buffer as item 5, and empty the buffer



Call procedure "COMM"

Reset the counter

Increment the counter

Test the counter

Note that the characters of ignorable items are not put in the character buffer, and the item is not reported. Also the action "read another character" occurs as the last action on every transition.

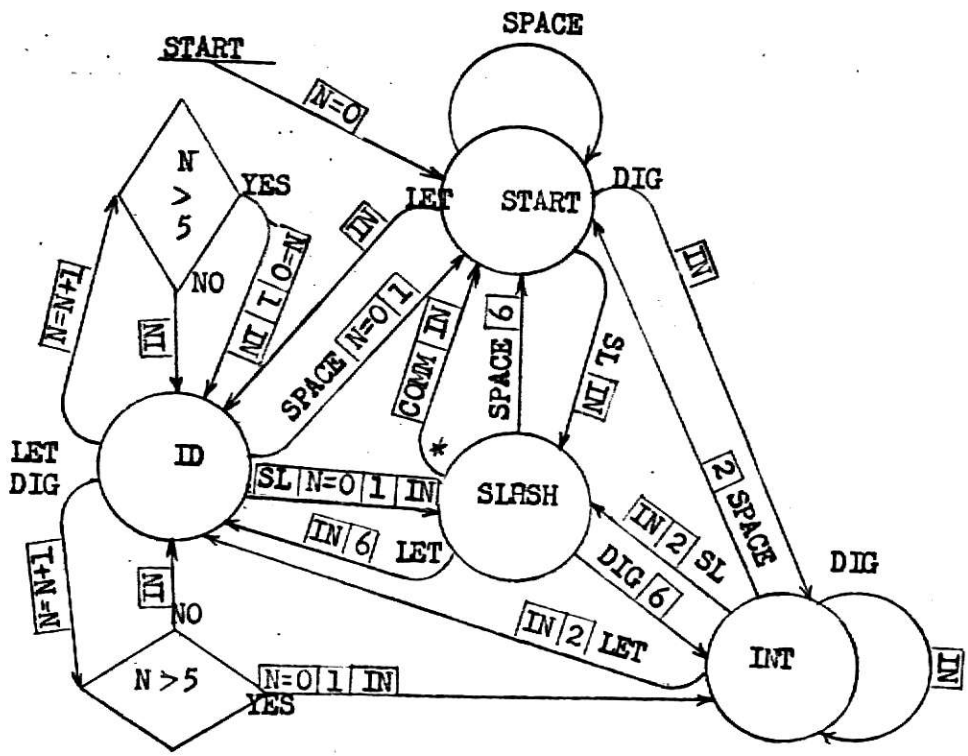


Figure 24. State diagram constructed by RWORD.

REFERENCES

1. Aho, A. V. and Ullman, J. D. The theory of Parsing Translation and Compiling, Vol. 1; Parsing. Prentice-Hall, Inc. Englewood Cliffs, N. J., 1972.
2. Cohen, S., The SPEAKEASY System. Argonne Physics Division Informal Report PHV-1972H. Argonne National Laboratory, Argonne, Illinois. (Feb. 1972)
3. Cohen, S. and Vincent, C. M. An Introduction to SPEAKEASY. Argonne Physics Division Informal Report PHY-1968E. Argonne National Laboratory, Argonne, Illinois (June 1972).
4. Feldman, J. A. A formal Semantics for Computer Languages and its Application in a Compiler-Compiler. Comm. ACM 1, Vol. 9 (Jan. 1966). pp. 3-9.
5. Gries, D. Compiler Construction for Digital Computers. John Wiley and Sons, Inc. New York, N. Y., 1971.
6. Johnson, W. L., Porter, J. H., Ackley, S. I. and Ross, D. T. Automatic Generation of Efficient Lexical Processors Using Finite-State Techniques. Comm. ACM 12, Vol. 11 (Dec. 1968), pp. 805-813.
7. Kain, R. Y. Automata Theory: Machines and Languages. McGraw-Hill, Inc., New York, N. Y., 1972.
8. Knobe, B. A Simple System for Generating Efficient Lexical Scans. Intern. J. Computer Math. Section A, Vol. 3, pp. 141-148.
9. Ross, D. T. Fourth-generation Software: A Building-block Science Replaces Hand-crafted Art. Computer Decisions (April 1970).
10. _____ The AED Approach to Generalized Computer Aided Design. Proc. ACM 22nd Nat. Conf., 1967, pp. 367-385.

EXTENSIBILITY OF LEXICAL ANALYZERS

by

GARY ANDERSON

B. S., Kansas State University, 1972

AN ABSTRACT OF A MASTERS REPORT

submitted in partial fulfillment of

the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas
1973

A tutorial report dealing with processors which select grammar primitives from a source language. The processors presented range from the most basic finite-state machines up to the most sophisticated analyzers capable of inserting synonyms, for changing the meaning of and inserting new language symbols. The analyzers discussed are based on the theory of finite-state machines and the proposal of adding semantic routines to these machines.

Two analyzer systems are presented. The first processor is AED-RWORD, "the industry standard", which was developed at M.I.T. as part of the AED-1 language translator. The RWORD system automatically generates lexical processors from descriptions of the language symbols given in terms of regular expressions. The second system presented is a lexical analyzer hand-coded from the proposed GENSCAN concepts. The GENSCAN concepts are a set of guidelines based on the theory of inserting semantic routines into a finite-state machine.

It is proposed that functions exist which allow the dynamic changing of the tables in such a manner as to change the meaning of symbols in the language being processed. These functions may be applied to either the RWORD lexical processor or to the processor constructed by GENSCAN concepts. These functions allow the language user the option of changing the meaning of symbols in order that the language is more meaningful to that individual.