A LEXICAL SCANNER GENERATOR FOR A MODULAR COMPILER GENERATION SYSTEM

by

TJEERD VENEMA

B.Sc., University of British Columbia, 1974.

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computer Science

We accept this thesis as conforming to the

required standard

The University of British Columbia

DECEMBER, 1975

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date Dec. 10/75

ABSTRACT

Much work has been done in the many aspects of compiler generation. We examine the problems associated with the generation of a full compiler and present a method of modular construction which would solve many of the problems which occur in previous generation systems. As an example of this modular construction, a lexical scanner generator is designed to produce lexical scanners which are easily interfaceable with the other components of a compiler.

ACKNOWLEDGMENTS

I would like to thank all those who, over a cup of coffee, were willing to argue with me about many little things, some of which grew into bigger things and found their way into this thesis. I would especially like to thank my supervisors, Harvey Abramson and John Peck, without whose help and guidance this thesis would not have been possible.

Chapter 1

1.1 The World of Compiler Generators

With the increase in the use of computers for both business and research applications, the problems involved in the efficient construction of a compiler for a given source computer language has become of interest to many areas of the computer industry. The dedicated mini-computer system is an example of this in that interaction with a dedicated system is best accomplished via a language which has been designed to provide a useful interface between the user and the system. This "useful interface" becomes increasingly essential as the degree of the user's understanding of the computer system decreases, something which is becoming more of a significant factor in today's world. For this reason, special purpose languages for such specific tasks as data base management are beginning to appear with increasing frequency.

In order to handle the increasing demand for new compilers and translators, the compiler writer requires increasingly sophisticated tools to assist in the task of generating a compiler or translator for a new or modified computer language. The ideal tool would be a method for the automatic production of a compiler from a design specification of the language for which the compiler is desired. A computer program which realizes this is the compiler generator, a program which takes as input a language specification and produces as output a compiler for the language.

1.2 The Block Compiler Generator

The earliest approach to compiler generation programs was to write the compiler generation system as a single program or "block". In this type of system, the language designer specifies certain properties of the language for which the compiler is desired; properties which the compiler generation system then uses to produce a compiler. The word "block" comes from the fact that the only user interaction with the system is through the specification of the initial input; beyond this, the user has no control over the generation process. Since a compiler is itself a complex program, the input specifications must be very long if they are going to describe the compiler completely. The problem with this is that long input specifications are both an invitation for user errors and result in large and unmodifiable compiler generators. All "block" compiler generators developed to date avoid this problem by taking the questionable approach of only allowing certain aspects of the final compiler to be specified. The Compiler-compiler developed by Jerome A. Feldman[1] in 1966 is an example of this type of compiler generation system. This system is best viewed via the diagram in Figure 1.

The Compiler-compiler first takes the formal syntax of a source language L, expressed in a syntactic meta-language (such as BNF) and feeds it into the syntax loader. This program builds the tables T1 which will control the parsing of programs in the source language L. Then the semantics of L, written in a semantic metalanguage, is fed into the semantic loader. This

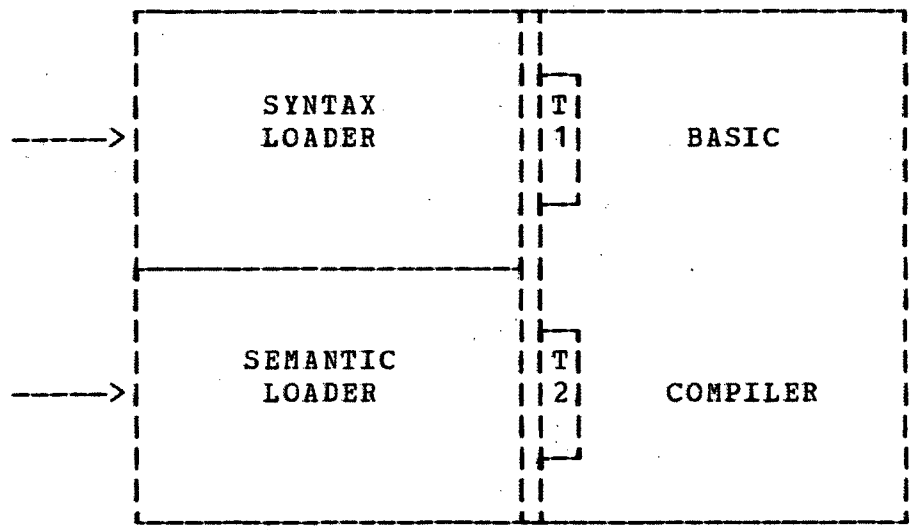


Figure 1

program builds another table T2, this one containing a description of the meaning of statements in L. When processing is completed, everything to the left of the double line is discarded, leaving a basic compiler for L.

At first glance, this system appears to solve the problem of compiler generation quite nicely. On closer observation, however, a few problems begin to show. The Compiler-compiler does not allow the user to supply a lexical specification of L (a description of what sequences of input characters are to be recognized as a single unit), it imposes upon the user the pre-defined lexical specification the Basic Compiler is capable of handling; one which has been included with the Basic Compiler by the implementor of the Compiler-compiler. Also, if the user discovers that a newly developed type of parser is more suitable to his needs, there is no method whereby the parser produced by the Compiler-compiler can be replaced by the newly developed

parser. A third problem is that the Basic Compiler is not extendable, a Basic Compiler for a language consists of a parsing section and a semantic interpretation section; but not all languages can be handled by a two phase compiler of this sort.

As an example of this, consider the language ALGOL-68[2]. The ALGOL-68 language definition specifies that the open ('(') and close (')') symbols can be used in many ways. They can be used to open and close closed-clauses (replace the symbols 'BEGIN' and 'END'), to open and close a conditional-clause (replace the symbols 'IF' and 'FI'), or to open and close a display; these being just some of the ways the open and close symbols can be used.

The open and close symbols can be used in so many ways that in some cases it is impossible to determine the meaning of an open symbol until the corresponding close symbol has been encountered. The problem is that a close symbol can be found arbitrarily far away from its corresponding open symbol. Thus, in order to determine the semantics of an open symbol, it may be necessary to look arbitrarily far ahead in the input stream, something which no conventional parsing system can do. In order to solve this probem, it is necessary either to have a sophisticated, non-conventional parser or to insert a special phase into the compiler between input and parsing which resolves any ambiguity surrounding the open and close symbols. Clearly, with the Compiler-compiler neither alternative is viable. The first alternative is not viable in that the Basic Compiler uses

only one type of parser, the one the system supplies; a parser which in order to achieve generality must have a conventional structure. The second alternative is also not viable as the Basic Compiler is a single unit; there is no allowance made in the Compiler-compiler for the user to specify that actions other than parsing or semantic interpretation are to occur as an intermediate phase. The Compiler-compiler is capable of compiler generation, but only if the language designer is willing to accept the restrictions the Compiler-compiler places on him as to the type and structure of the final compiler.

Another much more recent system is the Jossle[3] system developed at the University of California at Santa Barbara in 1973-74 which is diagrammed in Figure 2.

The syntax of the source language L, expressed in BNF, is fed into the grammar analysis program which produces tables to be used by an extended precedence parser. The semantics of L, expressed in the semantics language Jossle, is fed into the Jossle translator which produces semantic routines based on the rule reduction mechanism of the parser. A description of the lexical structure of the tokens is also fed into the system and a lexical scanner is generated as part of the parser. These two modules are then combined with two other modules, one for error handling and the other for code generation to form a compiler for the source language.

The Jossle system, being more recent, has been influenced by the theory of structured programming and as such the single block approach has been discarded in favour of a multi-block

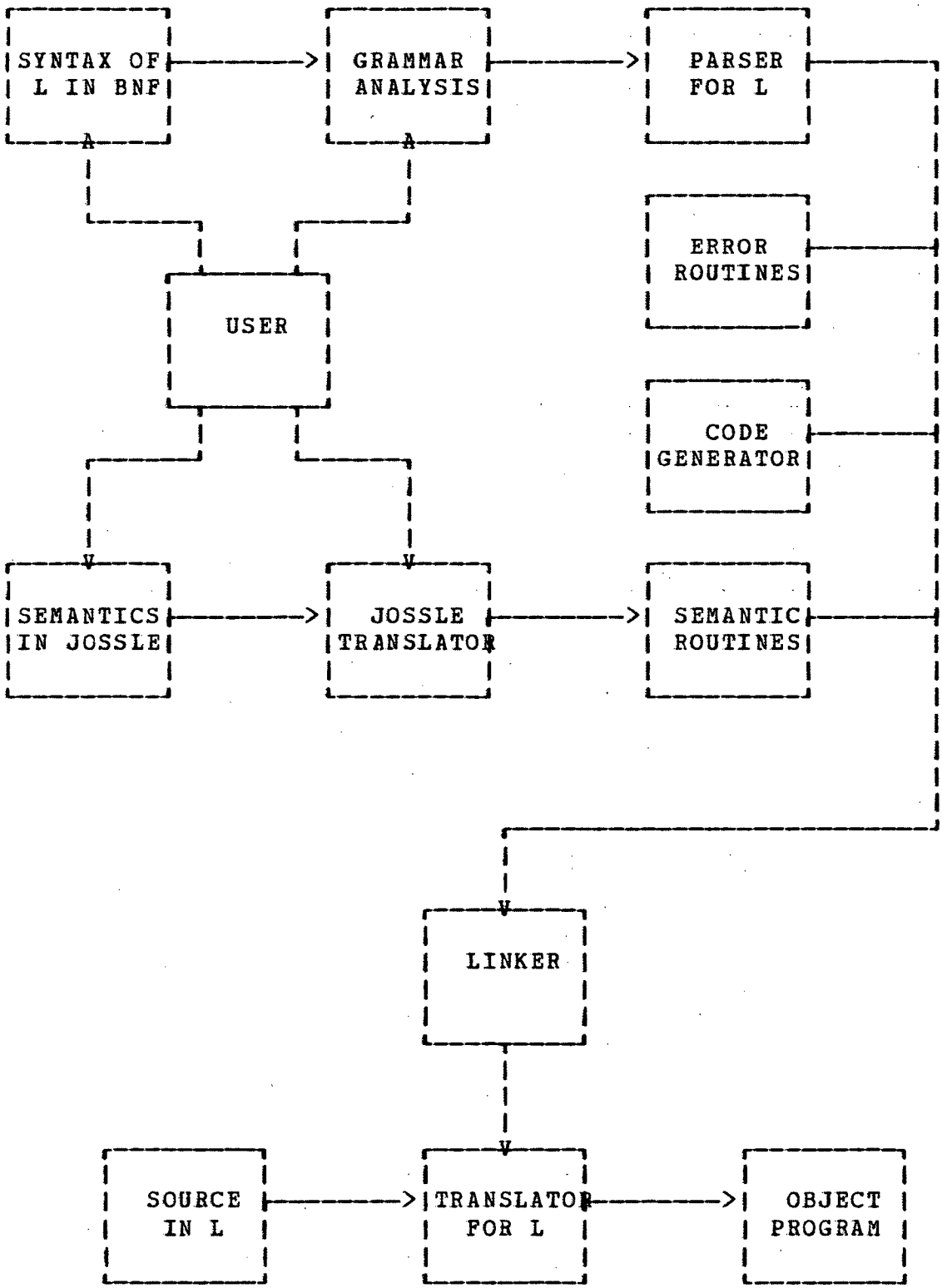

SYNTAX OF L IN BNF
GRAMMAR ANALYSIS
PARSER FOR L
ERROR ROUTINES
USER
CODE GENERATOR
SEMANTICS IN JOSSLE
JOSSLE TRANSLATOR
SEMANTIC ROUTINES
LINKER
SOURCE IN L
TRANSLATOR FOR L
OBJECT PROGRAM

Figure 2

system. The Jossle system recognizes the fact that a compiler is not a single unit and thus an easier way to generate a compiler is to handle each module separately and combine them as a last step. Should the supplied error routines prove lacking for some language, it is possible, although rather difficult, to modify that part of the compiler generator which generates the error module without disturbing the integrity of the entire system. This is still a rather inflexible approach in that it is necessary to modify the compiler generator physically if a change is desired. In the Jossle system, there would be nothing automatic about the generation of the new error routines; it would be the replacement of one set of hand coded routines with another. The same observation is true about the code generation routines.

Although the Jossle system has broken the compiler generation system down into four modules, the user only has control over two of them; the parser and the semantic routines. Even this control is limited in that when supplying the lexical specification for the lexical scanner which is generated as part of the parser module, it is only possible to define the appearance of items which the system has determined ahead of time it will recognize. It is possible to define the appearance of a comment, a string, an identifier, a real number, or an integer but if the source language contains two types of comments, more than one kind of string (i.e. one enclosed in apostrophes, the other in quotes) or allows identifiers with embedded blanks, then it is not possible to use the Jossle

system to generate a compiler for this language. The Jossle system has advanced from previous systems in that it recognizes the fact that compilers consist of separate modules, but it fails to recognize that how many modules or which modules should form the generated compiler is a decision which only the user can make. The modularity of a compiler should give a greater flexibility to the language designer, but he is only given a very restricted form of control over the specifications of the compiler which is produced.

1.3 The Building Block Compiler

What the user really wants from a compiler generation system is total control. If a compiler generation system which consists of one large mass of programs taking two, three, or four inputs chews viciously for a while and spits out a compiler about which the user knows very little due to changes and restrictions which have been placed on the compiler as the generation procedure progresses, then what is achieved is certainly not the compiler which was desired. In constructing a "block" system, the designer must make many decisions as he constructs the system, decisions which although correct for a large class of source languages can hardly be expected to be acceptable for all languages. Decisions such as the type of parser, type of error correction or method of semantic analysis are best left up to the user of the compiler generator.

In order to allow the user to make these decisions, it is necessary to break the large block of the traditional compiler

generation system down into many compiler component generation systems. The components produced by each of these component generators should conform to the components which are arrived at by breaking down the design of a compiler into logical components. For instance, a simple system which converts the stream input of a source language with reserved words into a tree structured representation could be described as in Figure 3.

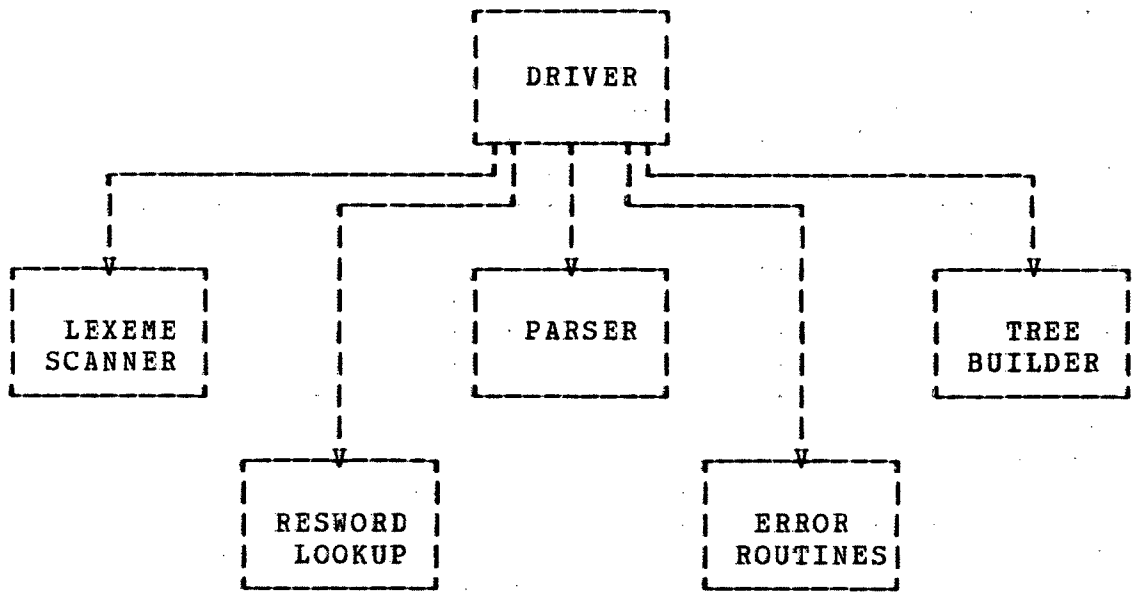


Figure 3

In this system, the driver program invokes the lexeme scanner which returns the next lexeme in the input stream. The driver may or may not choose to pass this lexeme on to the reserved word lookup for attempted identification as a reserved word in the source language. This step completed, the lexeme has been converted into a token, a terminal symbol for the

grammar on which the parser is based and as such is passed on to the parser for syntactic analysis. The parser could report that the token is in error, in which case the driver would invoke the error handling routines to attempt some form of error correction. On the other hand, the parser might return an indication that a reduction according to some rule of the source language grammar has occurred, in which case the driver would pass this information on to the tree building routine which would use this information in the construction of the tree representation of the input.

With this basic structure, if the source language does not have reserved words, then the reserved word identification module can be removed with minimal modification to the driver program and no modification to any other module.

Returning to the language ALGOL-68, both of the previously proposed solutions to the open and close symbol ambiguity could be applied: either the parser could be replaced with a more sophisticated version or an extra component could be inserted between reserved word identification and the parser giving a system of the form specified in Figure 4.

It is obvious that a "block" compiler generation system is unable to achieve this flexibility; a solution to this inflexibility is to have separate component generators for each of the logical components which make up a compiler and to allow the user to write his own driver program, a simple program which handles the interaction between the various generated components. Each of the compiler component generators would

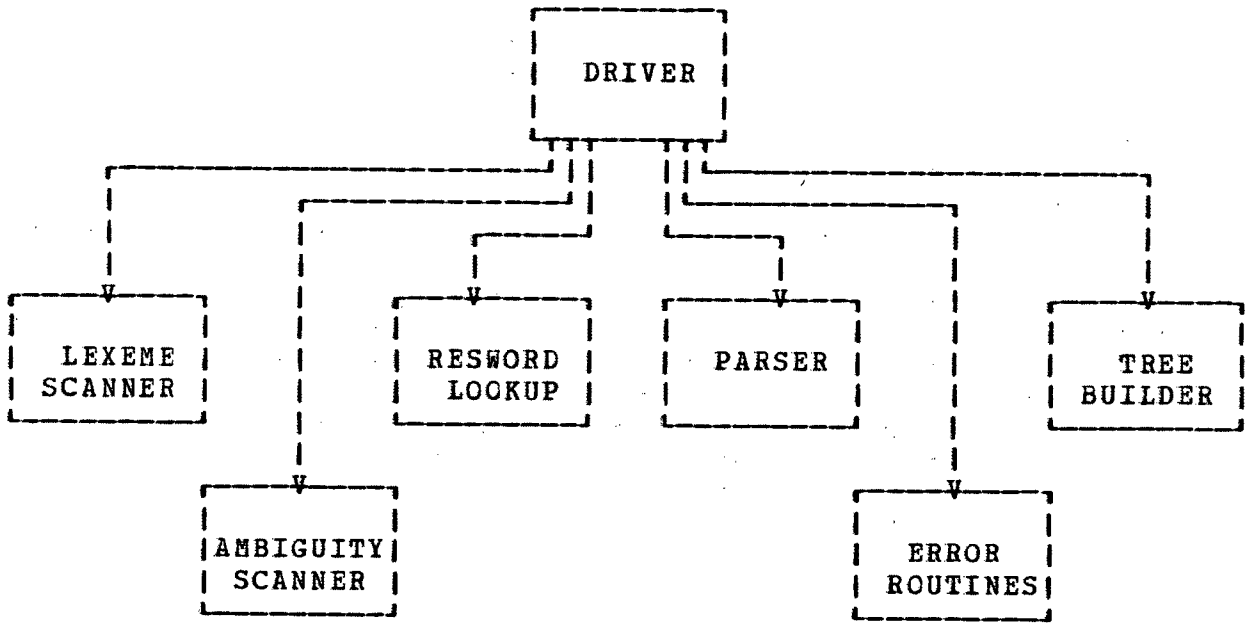


Figure 4

generate a component which performs a single logical task and is easily interfaceable, via the driver, with the rest of the system.

Much is known about the theory of one of the above components, the parser, and many parser generation systems exist which, with slight modification, would conform to the standards given above. The theory surrounding the other components is not as well developed, both error analysis and the handling of semantics are still matters of debate.

An example of a compiler component generator which can be used as part of a full compiler generation system is a lexical scanner generation system. Since some sort of lexical scan is necessary in any compiler, this component is required for all compilers and as such it is important that the component generator be flexible enough to handle the lexical description

of any language. On the other hand, the component generated should not contain code sections which are not required, a problem which is often encountered when a generation system is made "general". As an example of the generation of a component of a modular compiler generation system, a concentrated examination will be made of a lexical scanner generator.

Chapter 2

2.1 Design of a Lexeme Extraction Program

The first process which must occur in any compiler is the identification of symbols from the input stream. This process is basically an interfacing between the external world and the compiler itself. Thus a lexical scanner is a program which extracts from the input stream a sequence of symbols, combining them into lexemes. It is very important that a set of symbols which is to be recognized as a lexeme is recognized consistently as that same lexeme.

Definition 1

A *lexeme* is a sequence of symbols which are to be consistently recognized as a single unit from an input stream I over an alphabet A.

Definition 2

A *lexeme extraction program* is a program which breaks an input stream I over an alphabet A into lexemes.

Definition 3

A sequence of characters B over an alphabet A is a *prefix* of a sequence of characters S if S=BR where R is a (possibly empty) sequence of characters over A.

Figure 5 shows some examples of lexemes which could be recognized from some input stream.

Input	Lexeme
BEGIN X	BEGIN
THIS IS ONE :=	THISISONE
"XY""Z"	XY"Z

Figure 5

In the case of the second lexeme 'THISISONE' it should be noted that the input stream 'THIS ISONE :=' could also begin with 'THISISONE'. The extraction of a lexeme from the beginning of the input stream may be just a matter of recognizing n consecutive symbols and returning these n symbols, but it could also require a more complicated approach as is seen in the deletion of embedded blanks from within identifiers. A severe pitfall which occurs in many lexeme extraction programs in this regard is that if the lexeme extraction program is simple (for instance, does not allow deletion of blanks within identifiers) then there is a large class of languages for which this type of lexeme extraction program will not suffice. On the other hand, if the program is too complex (for instance, is responsible for the recognition of reserved words) then such a program is going to contain modules which are not always required. An examination of this shows that it is just a minor form of the problem associated with the "block" compiler generator; trying to accomplish too much at one time. Reverting to the logical

design of a compiler, an analysis of the two operations - recognizing a lexeme and identifying a reserved word - yields the observation that they are logically distinct and are best dealt with as separate components of a compiler generation system.

What is desired then is a lexeme extraction program which is powerful enough to handle the lexical specifications of almost all source languages and yet specific enough so that if the program is automatically generated it is understandable by the user and does not contain features the user does not want. This can be accomplished by the definition of a lexical machine.

2.2 The Lexical Machine

The basic problem in defining the operation of a lexical extraction program is a question of the amount of power which should be given to the program. Lexeme extraction programs have been written which, on close examination, turn out to be parsers whose action is to parse the input character by character, thus putting the lexeme together. The problem with this approach is that if these parsers are automatically generated, they are usually designed with lookahead in mind. A parser generated by such a system may look up to n symbols ahead (for some finite n). In order to generate a parser with up to n symbols of lookahead, it is necessary to make the parser complex enough to handle such lookahead, the result being that the lexeme extraction is rather slow. Also, it is not difficult to see that if a compiler uses one parser to do lexeme extraction and

another to do syntax analysis, there is a certain duplication of effort.

If, in order to recognize a lexeme, the lexeme extraction program needs to look farther than one character ahead in the input stream, then a modification should be made to the syntax specification of the language so that any lookahead of this sort is done by the syntax parser which, for any non-trivial language, is designed especially with this sort of lookahead in mind.

The problem with multi-character lookahead in a lexeme extraction program can be seen by defining a simple (and non-sensical) language LL as follows:

```
<PROGRAM> ::= IF ( <NUMBER> .EQ. <NUMBER> ) STOP
```

where the set of lexemes is defined as:

```
{IF,(,),.EQ.,STOP,<NUMBER>}
```

The lexeme <NUMBER>, in contrast to the other lexemes in the set, is not the sequence of characters '<NUMBER>' but is a more complicated lexeme which can be described as follows:

```
<NUMBER> ::= <SIGN> <INTEGER> |
             <SIGN> <INTEGER> . <INTEMPTY> <EXPONENT>
<INTEGER> ::= <DIGIT> | <INTEGER> <DIGIT>
<DIGIT> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<INTEMPTY> ::= <INTEGER> | empty

<EXPONENT> ::= E <SIGN> <INTEGER> | empty

<SIGN> ::= + | - | empty

This definition of the lexeme <NUMBER> describes the FORTRAN real and integer constants.

As the lexeme extraction program scans over an input stream such as '2.EQ.7)STOP', the following sequence of events occurs (the '|' represents the point at which the the lexeme extraction program is scanning):

|2.EQ.7)STOP

2|.EQ.7)STOP

At this point, the lexeme extraction program is unable to resolve its next action within one lookahead symbol (in this case, the '.'). If the program halts returning '2' as the next lexeme, then the parse will continue correctly. On the other hand, if the program continues to accept characters (trying to build a real constant) then the processing will continue with:

2|.EQ.7)STOP

2.|EQ.7)STOP

2.E|Q.7)STOP

At this point the lexeme extraction program would realize that too much of the input stream had been accepted since the string '2.E' is not a valid lexeme and the next symbol of the

input stream (in this case 'Q') cannot be accepted since no lexeme of the language has the string '2.EQ' as a prefix. The program would then have to "unwind" itself to discover which lexeme to return. Going back to the configuration:

2.|EQ.7)STOP

would give the valid lexeme '2.', But an error will certainly occur when an attempt is made to extract the next lexeme since the string 'EQ' is not a prefix of any lexeme in LL. The correct action would be to return to the configuration:

2|.EQ.7)STOP

It would require a complex lexeme extraction program to make these decisions correctly.

What this means is that to handle a language such as LL using the lexeme descriptions given requires the lexical extraction program to be a large, complicated system performing strange actions in order to extract the lexeme. This problem can be removed simply by not allowing lexemes such as <NUMBER> as part of the lexical specification of the source language. In the case of LL, this would mean modifying the syntax specification to:

<PROGRAM> ::= IF (<NUMBER> . EQ . <NUMBER>) STOP

<NUMBER> ::= <SIGN> <INTEGER> |

<SIGN> <INTEGER> . <INTEMPTY> <EXPONENT>

<INTEMPTY> ::= <INTEGER> | empty

<EXPONENT> ::= E <SIGN> <INTEGER>

<SIGN> ::= + | - | empty

and the set of lexemes to:

{IF,(,),.,EQ,STOP,<INTEGER>,E,+,-}

where <INTEGER> is defined as:

<INTEGER> ::= <DIGIT> | <INTEGER> <DIGIT>

<DIGIT> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The notion of a real number as a single lexeme has disappeared; a real number is now treated as a sequence of lexemes. For example, the string '3.14E-1' is no longer one lexeme, but a sequence of the lexemes '3', '.', '14', 'E', '-', '1'. Looking once again at the action of a lexeme extraction program on the string '2.EQ' the actions taken are:

|2.EQ.7) STOP

2|.EQ.7) STOP

At this point, there is no question as to the action which must be taken. The program returns '2' as a lexeme since this is the only thing it can do given the modified set of lexemes. The

lexeme extraction program for this modified version of LL is obviously simpler than the program which handles the original version of LL.

It should be noted that the above analysis does not mean that it is never possible to consider a real number as a lexeme, but that the choice of whether or not to represent a real number as one lexeme or several lexemes depends on the other lexemes being recognized. In the case above, if '.EQ.' was not a lexeme of LL, then the problem would not have occurred.

It is possible to modify the syntax and lexical specifications of any language in the above manner to the point where the lexeme extraction program requires no lookahead, however this in turn will mean that the syntax parser will be doing a character by character parse of the input stream, a very inefficient method for doing lexeme extraction. Certain abilities logically belong within the lexeme extraction program. For instance, many languages have within them facilities for handling the data type string, a string being defined as any sequence of symbols enclosed within quotes ('"'). Since it may be desirable to have the quote within the string itself, any quote within the string must be marked to indicate that it is just another character within the string and not the terminating quote. This is usually done by specifying that a quote within a string must be represented by two quotes ('""'); any occurrence of two quotes being interpreted as the occurrence of one quote within the string. For example:

"XYZ" represents the string XYZ

"XY""Z" represents the string XY"Z

From the viewpoint of the logical function of a lexeme extraction program, it should be able to handle this form of lexical analysis.

It will be shown that such an analysis can be done with one symbol lookahead, so that a lexeme extraction program with the ability to do one symbol lookahead will handle completely the logical notion of a lexeme. In order to define formally a system in which a lexeme extraction program has precisely the correct amount of power, it is useful to define the notion of a lexical machine, a machine which is driven by a lexical extraction program.

Definition 4

A simple lexical machine M is an ordered triple (H,A,B) where:

H - is a one character hold

A - is the lexeme part accumulated thus far

B - is the input stream not yet analyzed

together with the following machine instructions (where 'a' represents a single character):

1) ACCEPT - (empty,A,aB) => (empty,Aa,B)

2) IGNORE - (empty,A,aB) => (empty,A,B)

3) HOLD - (empty,A,aB) => (a,A,B)

4) ACCEPTHOLD - (h,A,B) => (empty,Ah,B)

5) IGNOREHOLD - (h,A,B) => (empty,A,B)

6) RETURN n - (empty,A,B) => (empty,empty,B)
and A is returned
together with n

7) ERROR - machine executes a fatal halt

The simple lexical machine is a machine which extracts the next lexeme from an input stream I. The machine has within it a 'hold', a location which can be used to hold the leading character of the input stream while looking at the next character. Since the HOLD instruction will only execute if the hold is empty, it is not possible to hold more than one character of the input stream thus making lookahead of more than one symbol impossible. Also, since the ACCEPT, and IGNORE instructions will only execute if the hold is empty, it follows that if a character has been placed in the hold, the next instruction must be one of ACCEPTHOLD or IGNOREHOLD. In this way it is impossible to add another character to 'A', the lexeme built thus far, without either discarding the character in the hold or adding it to 'A'. In this manner the lexical machine restricts the actions which are possible at any given time.

As an example, the sequence of instructions given in Figure 6 will extract the lexeme 'XY"Z' from the input stream '"XY""Z";'.

Diagramatically, the following happens with the input

```
IGNORE
ACCEPT
ACCEPT
HOLD
IGNOREHOLD
ACCEPT
ACCEPT
HOLD
IGNOREHOLD
RETURN 1
```

Figure 6

stream of '"XY""Z";'.

Step Number	Hold (H)	Lexeme (A)	Input Stream (B)	
1	H=	A=	B="XY""Z";	
2	H=	A=	B=XY""Z";	
3	H=	A=X	B=Y""Z";	
4	H=	A=XY	B=""Z";	
5	H="	A=XY	B="Z";	
6	H=	A=XY"	B="Z";	
7	H=	A=XY"	B=Z";	
8	H=	A=XY"Z	B=";	
9	H="	A=XY"Z	B=;	
10	H=	A=	B=;	and return XY"Z together with 1

The instruction set of the simple lexical machine allows sequences of instructions which will recognize many lexemes, however it is not able to handle all the lexemes which arise in various source languages. For example, consider again the language ALGOL-68 which has as part of its lexical specification the lexemes:

{+,:,+:=}

An input to a lexeme extraction program for Algol-68 could be the string '+:='. The best which can be done using the simple

lexical machine is the instruction:

ACCEPT

which leaves the machine in the configuration:

H= A=+ B=:=

Since the current value of A is a valid lexeme, the decision as to whether to return '+' or to accept the next character of the input stream (':') must be made. If the next instruction executed is a RETURN, then an incorrect analysis of the input stream will have been made. On the other hand, if the third character of the input stream is any character other than '=' (e.g. the input is '+=a'), then the correct action would be to return '+' as the lexeme. Since the action which the simple lexical machine must take after the first ACCEPT is dependent on the second character of the input stream and the simple lexical machine is designed to allow only single character lookahead, this situation represents an ambiguity for the simple lexical machine.

In this case, the problem occurred at the second character of the input stream, but it can occur arbitrarily far ahead when several lexemes overlap as they do here. A slight modification to the simple lexical machine will provide a solution to this problem.

Definition 5

A complex lexical machine CM is an ordered quadruple (m,H,A,B) where H, A and B are defined as in a simple lexical machine and m is a prefix of A. The instruction set of CM is that of M together with:

7) MARKTOKEN - (m,h,A,B) => (A,h,A,B)

8) BACKUPRETURN n - (m,h,mA,B) => (empty,empty,empty,AB) and m is returned together with n

A complex lexical machine is a simple lexical machine with the ability to mark the lexeme thus far built (the value of A) in one spot at any given time. At any time the decision can be made to either return the value of A (via a RETURN) or to return a prefix of A, the remainder of A being pushed back into the input stream. For the two sample inputs given above ('+:=' and '+:a'), the sequence of instructions given in Figure 7 displays the action which will occur.

Input '+:='	Input '+:a'
ACCEPT	ACCEPT
MARKTOKEN	MARKTOKEN
ACCEPT	ACCEPT
ACCEPT	BACKUPRETURN 2
RETURN 1	

Figure 7

Figure 7 also displays a property of the lexical machine, a

lack of control. In order to display the action of the complex lexical machine on two inputs, it is necessary to display two sets of instructions. Although the complex lexical machine, in contrast to the simple lexical machine, has an instruction set which resolves overlapping lexemes within one symbol lookahead, neither machine has the capability of breaking the sequential execution of instructions. This ability is provided by the definition of a controller.

Definition 6

A simple (complex) controller of a simple (complex) lexical machine is a machine which feeds a sequence of simple (complex) lexical machine instructions into a simple (complex) lexical machine.

The controller is responsible for determining what instructions are to be executed by either the simple or complex lexical machines. In the case of the inputs in Figure 7, it would be the responsibility of the controller to insure that the fourth instruction executed by the simple lexical machine was either an ACCEPT if the next symbol in the input stream was '=' or BACKUPRETURN otherwise.

2.3 The Design of a Lexical Scanner

There are many combinations of lexical machine instructions which could be put together by a controller and executed on a

lexical machine, however only some of these combinations actually represent desirable combinations in the creation of a lexeme extraction program. For instance, the sequence of simple lexical machine instructions:

HOLD
ACCEPT

will, by the definition of the simple lexical machine, cause a machine error when execution of the 'ACCEPT' instruction occurs since the hold is not empty. In a similar manner, it is possible to construct other sequences of instructions which will not execute on either the simple or complex lexical machines.

To insure that this does not happen a lexical scanner is defined. A lexical scanner is a controller which is defined in such a manner that any sequence of instructions produced by a lexical scanner for the lexical machine has the property that it will execute successfully when run on the lexical machine.

Definition 7 A simple lexical scanner L is a controller defined by the ordered sextuple (S,delta,F,E,E',s') over an alphabet A where:

S - is a set of states

F - is a subset of S, the set of final states

E - is the set of subsets of simple lexical machine instructions consisting of:

{ACCEPT}

{IGNORE}
{ACCEPTHOLD ACCEPT}
{ACCEPTHOLD IGNORE}
{IGNOREHOLD ACCEPT}
{IGNOREHOLD IGNORE}
{HOLD}
{ERROR}

E' - is the set of subsets of simple lexical machine instructions consisting of:

{RETURN n}
{IGNOREHOLD RETURN n}
{ACCEPTHOLD RETURN n}

delta - is the state transition function, delta = $delta^1$ U $delta^2$ where $delta^1$ and $delta^2$ are defined as the functional mappings:

$delta^1$: AxS => ExS

$delta^2$: A'xF => E'xs'

$A' = \{a \in A \mid f \in F \Rightarrow (a,f) \notin Domain(delta^1)\}$

s' - is an element of S, the start state

A simple lexical scanner is really a modified version of a finite state sequential machine. The simple lexical scanner moves from one state to another under the control of the function delta which determines the new state to enter as a function of the old state and the next input symbol. In a true finite state machine, this completely defines the purpose of the state transition function whereas in the case of a simple

lexical scanner, the state transition function in addition to controlling the state causes simple lexical machine instructions to be sent to the simple lexical machine for execution; instructions which may vary the contents of the input stream, the lexeme being built, or the hold of the simple lexical machine. This interaction is displayed in Figure 8.

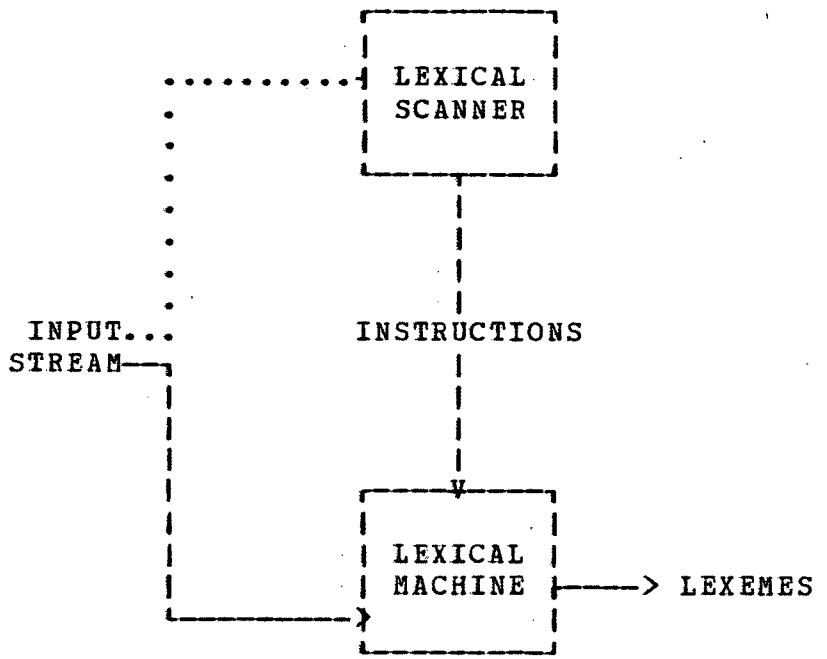


Figure 8

The main control of the lexical scanner is given by the function delta^1 which defines transitions between various states of the lexical scanner. In addition to this, if S $\in$ F (is a final state), then delta^2(S,a) might also be defined for some input character a. If S $\in$ F and delta^1(S,a) exists, then delta^2(S,a) cannot exist; but if delta^1(S,a) does not exist, then delta^2(S,a) may exist.

In terms of a lexical recognition system, the function

delta^1 is in control at any time when the lexeme thus far built (the value of A in the simple lexical machine) is not a valid lexeme. At any time when A is a valid lexeme (the lexical scanner being in state S with the head of the input stream being the character 'a'), then delta^1(S,a) might be defined in which case this transition must be taken (it is the only one since delta^2(S,a) cannot be defined if delta^1(S,a) is). If delta^1(S,a) is not defined, then it is possible that the transition delta^2(s,a) is defined in which case it is taken. Once a transition involving delta^2 has been taken, the next state is always the start state, and a lexeme is output from the simple lexical machine since the effect of a delta^2 transistion on the lexical machine is to cause execution of a sequence of simple lexical machine instructions, the last of which is always a 'RETURN n'.

A simple lexical scanner which would serve as a lexeme extraction program for a string constant as described previously can be defined as follows:

S - {s1,s2,s3,s4,s5}

F - {s5}

S' - {s3}

A - {a,b,c,...,z,"}

and delta is defined as:

State	Input	
	'	{A-'}
s1	(HOLD,s5)	(ACCEPT,s2)
s2	(HOLD,s5)	(ACCEPT,s2)
s3	(IGNORE,s1)	(ERROR,empty)
s4	(HOLD,s5)	(ACCEPT,s2)
s5	(IGNOREHOLD ACCEPT,s5)	(IGNOREHOLD RETURN 1,s3)

Under control of this lexical scanner, the simple lexical machine will undergo the following transitions when the input is '"XY""Z";'.

H	A	B	STATE	
empty	empty	"XY""Z";	s3	
empty	empty	XY""Z";	S1	
empty	X	Y""Z";	s2	
empty	XY	""Z";	s2	
"	XY	"Z";	s5	
empty	XY"	Z";	s4	
empty	XY"Z	";	s2	
"	XY"Z	;	s5	
empty	empty	;	s3	and returns XY"Z together with 1

In addition to a simple lexical scanner as a controller for a simple lexical machine, there is also a corresponding complex lexical controller for a complex lexical machine.

Definition 8

A complex lexical scanner LC is a controller defined by the sextuple (S,delta,F,EC,E',S') where S, delta, F, E' and S' are defined as in a simple lexical scanner and EC is defined as the union of E and the following set of subsets of the instructions of MC:

{BACKUPRETURN n}
{ACCEPT MARKTOKEN}
{IGNORE MARKTOKEN}

A complex lexical scanner is very much like a simple lexical scanner except that the complex lexical scanner is able to issue instructions which mark the lexeme being built and return only the lexeme up to the point of the mark if it so chooses.

The simple and complex lexical scanners are able to recognize almost all possible lexeme descriptions. An example of a case where a lexeme cannot be recognized by either a simple or complex lexical scanner is the FORTRAN Hollerith format code. This format code is specified in the form:

nnHcccc....c

where 'nn' represets an integer number and the 'nn' characters immediately following the 'H' are taken as a literal string to be printed on the output device. Such a lexeme is not recognizable by a simple or complex lexical scanner as it

requires the numerical value of 'nn' in order to extract the next part of the lexeme (or the next lexemes, if 'nn' 'H' and 'ccc...c' are considered to be separate lexemes) from the input stream. In order to do this, the lexeme extraction program must receive information as to the numeric value of 'nn' and use this information to modify itself so that it will pick up 'nn' characters following the 'H'. The next time a Hollerith format is encountered in the input, it will once again be necessary to extract a new value of 'nn' and use this new value in picking up the remainder of the format code.

Inherently this type of "dynamic" lexeme gives rise to several problems; designing a set of lexemes which require any lexeme extraction program to be self-modifying is undesirable and such lexemes can produce unusual results for the unwary user of such a program. Since the modification required to correct this difficulty is trivial (instead of

nnHccc...c

use

'ccc...c'

and the result, a sequence of characters between apostrophes, is now lexically simple) it seems to be a logical restriction not to allow the specification of lexemes which invoke such strange actions. It should be noted that the Hollerith format is also a source of problems to the users of the FORTRAN language in which it occurs for the very fact that the 'nn' characters after the

'H' are always taken as a literal string for output and if the user has miscounted (the inability to count above 1 is a property of those who deal with computers) strange results and error messages often appear.

Definition 9

A set of lexemes is lexically simple if it can be accepted by a simple lexical scanner.

Definition 10

A set of lexemes is lexically complex if it can be recognized by a complex lexical scanner, but cannot be recognized by a simple lexical scanner.

Venema's Postulate

If a set of lexemes is neither lexically simple nor lexically complex, then it is lexically ridiculous.

The postulate is rather ill defined; but it does portray the meaning which was intended. If a set of lexemes is constructed so as to be neither lexically simple nor lexically complex, then as has been shown with the Hollerith format, such a set of lexemes is logically questionable and is certain to cause problems to the user of the language for which the lexemes were designed. In this manner, ridiculous means a ridiculous choice of the representation of the lexemes. If the language designer has within his or her set of lexemes one or more

lexemes which is lexically ridiculous, then the designer can expect the usage of the language in question to drop a little for each lexically ridiculous lexeme used.

Chapter 3

3.1 Designing the Lexical Scanner Input

The simple and complex lexical scanners define a set of controllers which are able to act as lexeme extraction programs for a source language. As lexically complex languages, with a slight modification to the lexeme set, can be made lexically simple; the simple lexical scanner is adequate as a lexeme extraction program for virtually any source language. For this reason, the implementation of the lexical scanner generation system attempts only to generate simple lexical scanners for simple lexical machines.

A simple lexical scanner can act as a target for the lexical scanner generation program. This program takes as input a description of the lexemes which are to be extracted and outputs a simple lexical scanner which will do the extraction. The action of the simple lexical scanner on the simple lexical machine can then be simulated, effecting a lexeme extraction program. Although in many cases the simulation of abstract machines is a complicated process, the actions involved in both the simple lexical machine and a simple lexical scanner are so simple and suited to the task at hand that they can be simulated with no overhead due to the simulation. Because of this, the generated lexical scanner is able to execute at the same speed (not just the same order of magnitude) as a hand written lexeme extraction program for the same set of lexemes.

Since a simple lexical scanner is closely linked to a

finite state machine, it would seem desirable to allow the user to specify the input to the lexical scanner generator - a description of a set of lexemes - in terms of the finite state machine operations Kleene closure, catenation and union. This approach was taken in the development of the RWORD[4] system in 1960. With this system, the user describes each lexeme as an expression in terms of the three finite state operations. For instance, "a digit" would be expressed as:

```
DIGIT = 0 U 1 U 2 U 3 U 4 U 5 U 6 U 7 U 8 U 9
```

and a non-empty sequence of digits would be expressed as:

```
SEQDIGIT = DIGIT DIGIT
```

Using this method, the language designer is able to build any lexeme which can be described using regular expressions. Unfortunately, a lexeme extraction program based on regular expressions does not permit any form of lookahead and so recognition of the string '"XY""Z"' as the lexeme 'XY"Z' is not possible. The other problem with this approach is purely one of notation which assumes that the user of the RWORD system is familiar with the notation of regular expressions; which may not be true. Even for the user familiar with regular expressions, the regular expressions required to build up certain regular lexemes are complicated and quickly become unreadable. The RWORD system, although it does have the ability to ignore an

input character, does not have a mechanism whereby the user can specify that a character is to be ignored throughout the extraction of a particular lexeme. For instance, if a language has a lexeme for which internal blanks are deleted ('THIS IS ONE' becomes 'THISISONE'), then such a lexeme could not be expressed as a regular expression.

The lexical scanner is able to handle such situations in that it is not just a finite state machine, but a program which operates under finite state control, in turn controlling a simple lexical machine which does the actual lexeme extraction. It is the simple lexical machine which gives the extra power needed to recognize complicated, but well defined lexemes. The use of regular expressions for describing the lexeme set to the lexical scanner generation program is not general enough to make full use of the power of a lexical machine.

The type of description for the input to the generation system which is wanted is one in which the user feels comfortable, one in which he is not required to learn extra facts, but in which he can express himself naturally. The question of what input specification to use then becomes one of determining what comes naturally to the user (or for that matter, to anyone at all). Taking as an example the notion of an identifer (a lexeme common to most languages), it could be described completely by the following BNF syntax desciption:

<IDENTIFIER> ::= <LETTER> <TAIL>

<TAIL> ::= <LETTER> <TAIL> | <DIGIT> <TAIL> | empty

```
<LETTER> ::= A | B | C | D | E | F | G | H | I | J |
             K | L | M | N | O | P | Q | R | S | T |
             U | V | W | X | Y | Z
<DIGIT> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

This description describes completely an identifier, but the entire syntax can be expressed in English as "a letter followed by any sequence, possibly empty, of letters or digits". Although English is often ambiguous, in this case the English description portrays the exact meaning of what is wanted and is more easily understood. The lexical scanner generation system attempts to take this into account, and uses a description of the lexemes which closely parallels the English form given above.

3.2 The Input Language

The input to the lexical scanner generation system is a description of the ways in which several pre-defined basic units are assembled. These basic units are described as follows:

ONE OF "charseq"

The value of 'charseq' is any sequence of characters and the next character of the input stream must be a character in 'charseq'. The next character of the input stream is added to the lexeme being built.

ANY OF "charseq"

The value of 'charseq' is any sequence of characters. As long as the next character of the input stream is a character in 'charseq' it is added to the lexeme being built. The statement ANY OF includes the possibility that there may be none at all.

NONE OF "charseq"

The value of 'charseq' is any sequence of characters and the next character of the input stream must not be a character in 'charseq'. The next character of the input stream is added to the lexeme being built.

NOTANY OF "charseq"

The value of 'charseq' is any sequence of characters. As long as the next character of the input stream is not a character in 'charseq' it is added to the lexeme being built. The statement NOTANY OF includes the possibility that there may be none at all.

IGNORE "charseq"

The value of 'charseq' is any sequence of characters and the next character of the input stream must be one of 'charseq'. The next character of the input stream is discarded.

"charseq"

The value of 'charseq' is any sequence of characters. If B is the input stream, then the value of 'charseq' must be a

prefix of B. 'charseq' is added to the lexeme being built and deleted from the head of the input stream.

These six basic units form the basis for building more complex lexeme descriptions by catenation, using the ',' as a catenation operator. The description, using these primitives, of the identifier defined above is:

```
ONE OF "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        ANY OF "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
```

which corresponds very closely to the English description which was "a letter followed by any sequence, possibly empty, of letters or digits". A series of basic units catenated by ','s is known as a sequence.

There are two minor problems which arise in the use of the 'charseq' representation in the above structures. The first is that since 'charseq' can be any sequence of characters, the delimiting character (the quote '"') can be one of the characters of 'charseq'. The solution used by the lexical scanner generator is the standard one for strings with a single delimiting character, if one of the characters within 'charseq' is to be a quote, then doubling it causes it to appear once within the string. The second problem is what to do about characters which are required as part of a lexeme but which cannot be used since the device being used to input the desciptions does not have these characters. This problem is

solved by allowing the user to specify between apostrophes the decimal value of the character to be recognized. Thus:

"AB'1'C"

used as a basic unit specifies that the input stream must contain an 'A', followed by a 'B', followed by the character which has an internal numeric value of one, followed by a 'C'. Because of this ability, if an apostrophe is desired as one of the characters in 'charseq', it is necessary to use two apostrophes. Thus:

"AB''C"

used as a basic unit specifies that the input stream must consist of an 'A' followed by a 'B' followed by a ''' followed by a 'C'.

This ability is also useful for the detection of certain "event" lexemes. An event lexeme is not a true lexeme, but is a method whereby a lexeme extraction program is able to report the occurrence of events such as the end of a source line or source file. This capability is needed in languages such as FORTRAN or BCPL[5] where knowledge of the end of source line is required in order to continue syntactic analysis. In the lexical scanner generation system, it is possible to write the routine responsible for the reading of source lines in such a manner that the routine appends to each line a (machine dependent)

character which cannot (or is highly unlikely to) occur anywhere in the input stream (e.g. the character which has an internal numeric value of one on an IBM 370). This single character can then be defined as a lexeme, and every time this lexeme is returned is an indication that the end of a source line has been encountered.

Occasionally it is useful to be able to give two specifications for the same lexeme. For instance, a language might have two forms of a comment, one of which is the word 'COMMENT' followed by any sequence of symbols not including a semi-colon (;) and terminated by a semi-colon, the other a per-cent ('%') sign followed by any sequence of characters not including a per-cent sign and terminated by another per-cent sign. The first form of the comment can be expessed as:

```
"COMMENT", NOTANY OF ";", ";"
```

and the second form can be expressed as:

```
"%", NOTANY OF "%", "%"
```

The user thinks of a comment as being "the first description or the second description", and the input notation for the lexical scanner generator reflects this thinking by allowing this to be expressed as:

```
"COMMENT", NOTANY OF ";", ";" OR "%", NOTANY OF "%", "%"
```

where the use of 'OR' means either the first sequence or the second sequence. Since many users of such a system for generating lexeme extraction programs are also familiar with BNF notation, the word 'OR' can be replaced by the symbol '|' as in:

```
"COMMENT", NOTANY OF ";", ";" | "%", NOTANY OF "%", "%"
```

It is possible to add further alternatives by adding the word 'OR' followed by another sequence as often as is needed. A series of sequences separated by 'OR's is known as a section.

The basic units can also be built up into more complex structures by the use of the statement:

```
id IS section.
```

where 'id' is an identifier which is to be associated with the name 'id'. Thus:

```
subchar IS NOTANY OF """" OR IGNORE """", """".
```

associates with the name 'subchar' part of a lexeme description for a string as defined previously. Any string which is enclosed within quotes can have the character sequence between the enclosing quotes broken down into pieces, each of which is either a sequence of characters without quotes or two quotes the first of which is ignored.

Once a name has been associated with a section, it is

possible to use that name in any one of the following statements.

ONE OF id

This is equivalent to 'section' where 'section' is the section which has been associated with the name 'id'.

ANY OF id

This is equivalent to 'section, section, ...' sequenced as many times (including zero) as is required to continue accepting characters from the input stream where 'section' is the section associated with the name 'id'.

NOTONE OF id

If 'section' is the section associated with the name 'id', then 'NOTONE OF id' is equivalent to 'not-section' where 'not-section' is derived from 'section' by reducing 'section' to its basic units and altering each occurrence of 'charseq' to {A - 'charseq'} where A is the alphabet consisting of those characters with an internal decimal representation n, 0<=n<=255.

NOTANY OF id

If 'section' is the section associated with the name 'id', then 'NOTANY OF id' is equivalent to 'not-section, not-section, ...,' sequenced as many times (including zero) as is required to continue accepting characters from the input stream where 'not-section' is derived from 'section' by reducing 'section' to

its basic units and altering each occurrence of 'charseq' to {A - 'charseq'} where A is the alphabet consisting of those characters with an internal decimal representation n, 0<=n<=255.

IGNORE id

If 'section' is the section associated with 'id', then 'IGNORE id' is equivalent to 'ig-section' where 'ig-section' is derived from 'section' by reducing 'section' to its basic units and causing all transitions of 'section' to discard any characters normally accepted.

Any occurrence of the above five primitives is again a sequence, thus the definition system is recursive.

Since many parsers require that a number be associated with each lexeme, it is useful to be able to specify to the lexical scanner generator to associate with each lexeme a number which the lexical scanner is to return if that lexeme is encountered. In this manner, the lexical scanner can return both the lexeme detected as well as the number associated with that lexeme. The lexical scanner generator allows this by providing a statement of the form:

LEXEME number IS section.

where 'section' is as specified above. The value of 'number' could be an integer, for example:

```
LEXEME 7 IS section.
```

or it could be an identifier:

```
LEXEME thisone IS section.
```

In which case the identifier 'thisone' must be associated with an integer value via a statement of the form:

```
thisone := integer.
```

which must occur before the statement 'LEXEME number IS section.' is encountered ('integer' being an integer constant).

In addition to any of the above specifications, it is possible to include the following two primitives anywhere in a sequence as a basic unit.

NULL "charseq"

The value of 'charseq' is any sequence of characters which are to be treated as null-characters (hence ignored if they occur at the head of the input stream) from this point till the end of the lexeme description. For example:

```
LEXEME 9 IS NULL " ", ONE OF "A", ANY OF "ABCDEF".
```

will accept any sequence of the characters "ABCDEF" which begins with an "A", ignoring all blanks encountered while building the

character sequence.

NOTNULL "charseq"

The value of 'charseq' is any sequence of characters. If any of 'charseq' is being treated as a null character, from this point till the end of the lexeme description it will again be recognized as a regular character.

A sequence of desciptions of this form then make up the total input to the lexical scanner generator. For syntactic reasons, the sequence of descriptions must be enclosed between the symbols 'BEGIN' and 'END'.

Returning to the problem of recognizing the lexeme 'XY"Z' from the input '"XY""Z"' as a string, the representation of the lexeme 'string' would be given as:

```
BEGIN
   SUBCHAR IS NOTANY OF """", IGNORE """", """".
   STRING := 1.
   LEXEME STRING IS IGNORE """", ANY OF SUBCHAR, IGNORE """".
END
```

3.3 Implementation of the Lexical Scanner Generator

The program (written in PL360[6]) which does the lexical scanner generation takes as input a description of a set of lexemes in the form described above. As output, it produces either a program in a "description language" describing the operation of the lexical scanner or a program in the language

PL360 which can be compiled into an object program which, when run, effects the lexical scanner. The main purpose of using the description language is to enable the user to check the operation of the final lexical scanner to verify that the lexical scanner produced reflects the expectations as to what lexemes were specified.

3.3.1 The Description Language

The description language is a readable method of expressing the state transition function (delta) of the simple lexical scanner. There are four forms of instructions which affect control flow within the description language (anything within braces is optional).

{Sn} IF "charseq" THEN (instr-seq-1) {ELSE (instr-seq-2)}

The value of 'n' if 'Sn' is present is an integer, in which case 'Sn' is a state label. If the next character of the input stream is any one of 'charseq' then the sequence of instructions 'instr-seq-1' is to be executed, otherwise the sequence of instructions 'instr-seq-2' is to be executed. The values of 'instr-seq-1' and 'instr-seq-2' may be any element of the set E as defined in the simple lexical scanner (with the same meaning) together with the instruction 'GO Sn' where 'Sn' is a state label which transfers control to the instruction labelled 'Sn'.

{Sn} IFNOT "charseq" THEN (instr-seq-1) {ELSE (instr-seq-2)}

The value of 'n' if 'Sn' is present is an integer, in which

case 'Sn' is a state label. If the next character of the input stream is not any one of 'charseq' then the sequence of instructions 'instr-seq-1' is to be executed, otherwise the sequence of instructions 'instr-seq-2' is to be executed. The values of 'instr-seq-1' and 'instr-seq-2' may be any element of the set E as defined in the simple lexical scanner (with the same meaning) together with the instruction 'GO Sn' where 'Sn' is a state label which transfers control to the instruction labelled 'Sn'.

{Sn} WHILE "charseq" DO (instr-seq)

The value of 'n' if 'Sn' is present is an integer, in which case 'Sn' is a state label. As long as the next character of the input stream is any one of 'charseq' then the sequence of instructions 'instr-seq' is to be executed. The value of 'instr-seq' may be any element of the set E as defined in the simple lexical scanner (with the same meaning).

{Sn} WHILENOT "charseq" DO (instr-seq)

The value of 'n' if 'Sn' is present is an integer, in which case 'Sn' is a state label. As long as the next character of the input stream is not any one of 'charseq' then the sequence of instructions 'instr-seq' is to be executed. The value of 'instr-seq' may be any element of the set E as defined in the simple lexical scanner (with the same meaning).

In addition to the four control instructions as defined

above, the fifth and final instruction in the description language is:

(instr-seq)

The sequence of instructions 'instr-seq' is to be executed. The value of 'instr-seq' may be any element of the set E as defined in the simple lexical scanner (with the same meaning).

The lexical scanner generator can also produce a source program for the language PL360 to enable the user to get a working version of the lexical scanner. The PL360 source code generated is machine dependent in that the language PL360 is especially designed for the IBM 360 and 370 series computers and so PL360 compilers do not exist for most other computers. If the resulting lexical scanner is meant to run on a computer other than the IBM 360 or 370, then it is necessary to write a translator for the machine independent description language to a language for the computer on which the lexical scanner is to run.

The purpose of the lexical scanner generator is to translate from the input language to the description language. The analysis done to accomplish this is similar to that done for finite state machines. For example, the following is a legal input to the system.

```
BEGIN
   LEXEME 1 IS ":".
   LEXEME 2 IS ":=".
```

```
END
```

which produces as instructions in the description language:

```
S1   IF ":" THEN (ACCEPT GOTO S2) ELSE (ERROR)
S2   IF "=" THEN (ACCEPT RETURN 2)
     (RETURN 1)
```

If each lexeme is regarded as a sequence of transitions, then there is the problem of what to do when a ':' appears at the beginning of the input stream since a ':' could be the start of either lexeme 1 or lexeme 2; a non-determinism that must be resolved. This is similar to the same situation arising in the case of finite-state machines.

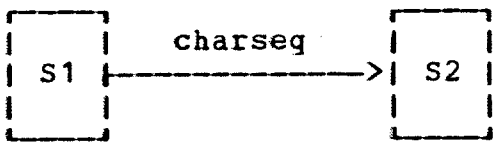
Since the lexical scanner is similar to a finite state machine, the program is broken down into seven phases resembling operations on finite state machines: the construction of an initial finite state machine with empty transitions (NDFSMWET), the construction of an empty transition graph (ETG), the construction of the transitive closure of the empty transition graph (TCETG), the construction of the non-deterministic finite state machine (NDFSM), the construction of the deterministic finite state machine (DFSM), the analysis to check for correctness of the DFSM, and the punching of the final machine in the description language. Although each of these phases bears the name of its corresponding transformation with finite state machines, the phases are not completely the same as the

corresponding transformations since the effects on the simple lexical machine, which must be considered in each of the seven phases, do not enter into the corresponding finite state machine transformations.

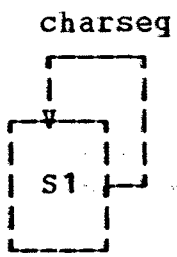
3.3.2 The Initial Machine (NDFSMWET)

The first phase in the generation of a lexical scanner is the translation of the input to an internal representation. This is done by creating an internal representation of the following form for each of the input primitives.

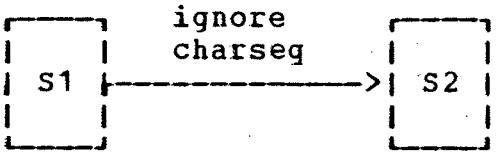
ONE OF "charseq"



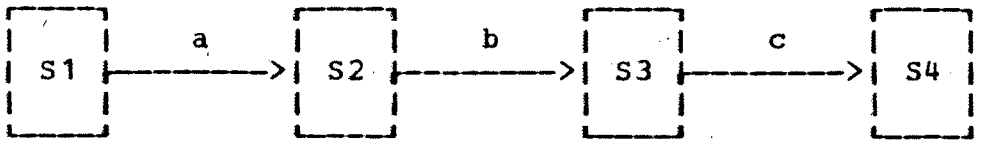
ANY OF "charseq"



IGNORE "charseq"



"charseq" (e.g. "abc")



For the first two, the negation forms ('NOTONE' for 'ONE', 'NOTANY' for 'ANY') are created by creating a structure for 'ONE OF {A - charseq }' for 'NOTONE OF "charseq"' and 'ANY OF {A - "charseq"}' for 'NOTANY OF "charseq"'; where A is the alphabet consisting of the characters whose internal representations have the numerical values 0 to 255. In addition, if the current set of NULLed characters (those characters being ignored as the result of a 'NULL "charseq"' having been encountered) is not empty, than to each arc in any of the above is added the labelling

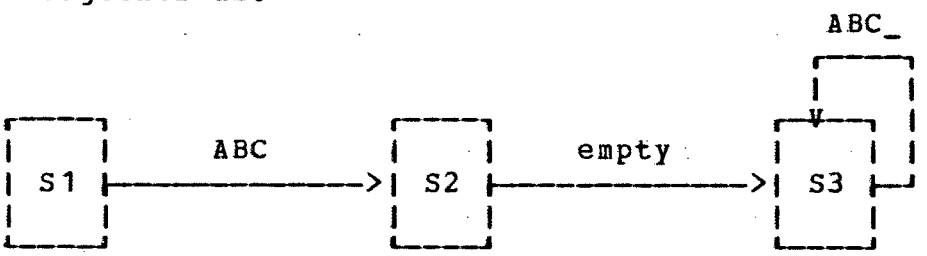
ignore nullchars

The "glue" which is used to build the lexeme descriptions

from these simple building blocks is the empty transition. Thus, the lexeme description

ONE OF "ABC", ANY OF "ABC_"

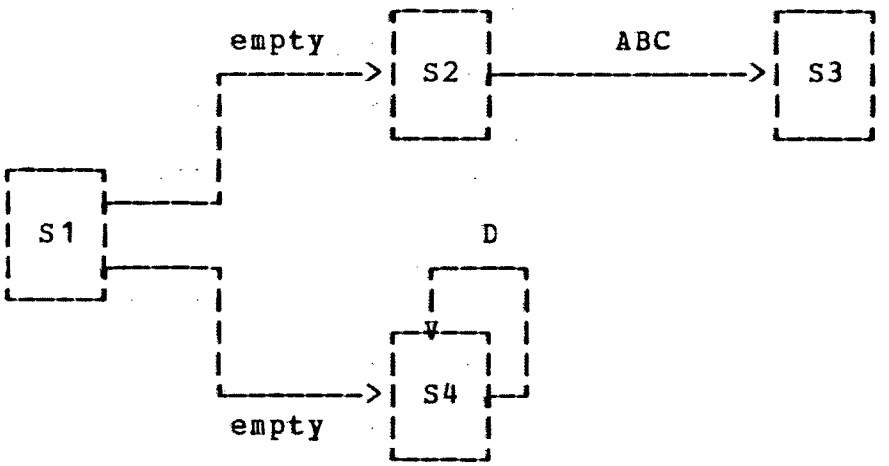
is put together as:



and the lexeme description:

ONE OF "ABC" OR ANY OF "D"

is put together as:



In addition to this, one state, the start state, is built which has an empty transition to the first state of the internal

representation of each of the lexeme descriptions. With the exception of the notion of transitions being labelled 'accept' (retain the character at the head of the input stream) or 'ignore' (discard the character at the head of the input stream), the internal structure built is a finite state machine.

3.2.2 The Empty Transition Graph

Once the NDFSMWET has been created, it is necessary to remove the empty transitions before attempting any form of algorithm for converting from a non-deterministic to a deterministic form of the machine. In order to achieve this, it is necessary to know, for any two states s1 and s2 if it is possible to reach s2 from s1 via an empty transition path. This information is obtained by first constructing an empty transition matrix M, a binary matrix for which M(i,j) is 1 if and only if there is an empty transition from state si to sj and then finding the transitive closure M' of M. The matrix M is constructed by first setting M to zero and then setting M(i,j) to 1 if an empty transition exists between state si and sj.

3.2.3 The Transitive Closure of the Empty Transition Graph

In order to determine not only if state s1 has an empty transition to state s2, but if state s2 can be reached from s1 by some sequence of empty transitions, it is necessary to compute the transitive closure of the ETG. The algorithm used to do this calculation is a modification of the Warshall[7] algorithm developed by Warren[8]. In this algorithm, the fact

that the system is implemented on an IBM 370 with both a Translate and Test and an Or Character instruction is taken into account. Also, this algorithm requires that only one pass be made over the matrix. If it is possible to test w entries of M at a time for an all-zero value and to "or" v entries at a time (both v and w are 255 in this implementation), and M is an nxn matrix, then for large n, the limiting cases give n^2 to n^3/v operations for Warshall's algorithm and n^2/w to n^3/v for Warren's algorithm. The fact that M is reasonably sparse means that if n is the number of states, the algorithm requires approximately $O(n^2)/255$ operations rather than the $O(n^2)$ operations required by the Warshall algorithm using the Translate and Test and Or Character instructions, or the $O(n^3)$ operations required by the standard Warshall algorithm. This is an important fact since the number of states created in the translation of the input could be quite large; initial timings for an ALGOL-W[9] lexical scanner have indicated that the system spends 90 to 95 per-cent of its time in this phase if the standard Warshall algorithm is used and only 10 per-cent of its time if the Warren algorithm is used.

3.2.4 Computation of the NDFSM

With the computation of the transitive closure of the ETG comes the necessary information to form the NDFSM. The method used is the standard method for removing transitions from a NDFSMWET which can be outlined as follows:

> For each state Si, if M(i,j)=1 then add all the non-empty transitions of state Sj to Si. If Sj is final, then mark Si as final. After this process has been carried out on all the states in the system, those states which can no longer be reached from the start state may be discarded.

The proof of the correctness of this algorithm is given by Aho, Hopcroft and Ullman[10]. The algorithm as it stands works for the NDFSWET even though the NDFSM is not really a finite state machine provided the information to accept or ignore the input character is carried along with a transition when that transition is added to another state.

3.2.5 Computation of the DFSM

With the assurance that the NDFSM has no empty transitions, it is possible to remove non-deterministic transitions. It is again necessary to consider the fact that the NDFSM is not quite a finite state machine in that the transitions are labelled either 'string' (accept the character of the transition) or 'ignore' (ignore the character of the transition). In the removal of non-deterministic transitions in a finite state machine F, a new machine F' is created whose states are the subsets of the states of F. If F contains a state S which has a transition on a character a to each of n states s1, s2, ..., Sn then F' has a transition from {S} to {s1,s2,...,sn} on a. The effect of this is then to "collapse" the n transitions of F to

one transition in F'.

In the NDFSM, this is not quite so simple in that in addition to the transitions of the NDFSM being labelled by the character of the transition, they are also labelled either 'string' or 'ignore'. From the meaning of 'string' and 'ignore' given above, it is easy to see that if a set of n transitions from si to sj on a character a are all labelled 'string' then the resulting transition in the new machine should be labelled 'string'. The same is true if all the transitions are labelled 'ignore', the problem being in the case where some of the transitions are labelled 'string' and some are labelled 'ignore'.

In order to handle this, a new type of transition is created, the 'limbo' transition. If a transition is labelled 'limbo', then this transition has been created by "collapsing" a number of transitions, some of which were originally labelled 'string' and some of which were originally labelled 'ignore'. Aside from this labelling change, the NDFSM can be treated as if it were a non-deterministic finite state machine and the standard power-setting algorithms can be used.

In order to determine which subset of the states of the NDFSM is to be the state of the DFSM to which a transition is to go, it is necessary to examine all possible combinations of the transitions of each particular state of the NDFSM to determine if the intersection of the characters labelling each transition in the combination is empty or not. If it is not empty, there is a non-determinism between those transitions which make up the

combination being examined. Although the combinations could be looked at in any order, for reasons of efficiency it is better to examine the large combinations before the small ones. For example, suppose state s1 has the following transitions:

to state s2 on a,b,c
to state s3 on a
to state s4 on a,b

If the first combination examined was the combination consisting of the first two transitions, then the non-determinism in the character 'a' would be detected, giving a new set of transitions:

to state {s2,s3} on a
to state s2 on b,c
to state s4 on a,b

The non-determinism between states {S2,S3} and S4 would then be detected later, but if the combinations had been examined in a different order, from the largest combination to the smallest, then it is obvious that it would never be necessary to include any newly created states in the powersetting process. In the above example, if the first combination of transitions examined was the combination consisting of all three transitions, then the resulting transitions after analysis would be:

```
to state {s2,s3,s4} on a
to state s2 on b,c
to state s4 on b
```

The three combinations involving two transitions would then be examined giving the deterministic transitions:

```
to state {s2,s3,s4} on a
to state {s2,s4} on b
to state {s2} on c
```

If c is the (approximately constant) number of operations required to examine one combination of transitions, then in order to examine all the combinations of the n transitions of a given state it is necessary to perform O(c2**n) operations. Since the number of operations involved in keeping track of which transitions are currently being examined tends to make c rather large, the power-setting process becomes very slow for large n.

A quick examination shows that the process of creating the NDFSM from the NDFSMWET can only increase, never decrease, the number of transitions out of a given state. Since, by the rules of the construction of the NDFSMWET the start state of the system must initially have m empty transitions where m is the number of lexemes being defined, it follows that the same state in the NDFSM must have at least m transitions. Since for many languages m is in the range of 25 to 50, this means that the

time required to power-set the start state is $O(c2^{30})$ or approximately 1,000,000,000. Thus it becomes impractical (for 30 lexemes, approximately 20 minutes on the IBM 370) to power set the start state.

The example mentioned above is really a worst case for power-setting in that the character 'a' occurs in each transition. In many cases (and for the start state of the NDFSM, most cases) it is possible to break down the set of transitions T for a particular state into subsets T1, T2, ..., Tn such that:

$$\{\{a \mid a \text{ labels } Ti\} \text{ intersect } \{a \mid a \text{ labels } Tj\}\}=\emptyset \ \forall i,j$$

Definition 11

If T is the set of transitions of a state S, then P(T) is the set of transitions obtained after power-setting T.

Theorem

Let S be a state, T be the set of transitions {t1,t2,...,tn} out of S and Ai be the set of characters labelling ti ($1<=i<=n$).

If there exist T1, T2, ..., Tk subsets of T such that $T=T1 \cup T2 \cup \ldots \cup Tk \ \& \ \forall r \neq s \ ti \in Tr, tj \in Ts \Rightarrow \{Ai \text{ intersect } Aj\}=\emptyset$

then

$$P(T)=P(T1) \cup P(T2) \cup \ldots \cup P(Tk)$$

Proof

If suffices to show that if delta is the state transition function defining T, delta' the state transition function defining P(T) and delta'' the state transition function defining P(T1) U P(T2) U ... U P(Tk) then delta'=delta''.

Suppose the transitions out of S on a are:

t1 is delta(a,S)=S1
t2 is delta(a,S)=S2
.
.
tm is delta(a,S)=Sm

then by definition of powersetting, P(T) yields:

$$delta'(a,S)=\{S1,S2,...,Sm\}$$

and this is the only transition on (a,S). By the definition of Ti, {t1,t2,...tm} is a subset of Ti for only one i since if {t1,t2,...,tm} is a subset of two transition sets Ti and Tj then a $\in$ Ai, a $\in$ Aj (by definition of Ai, Aj) and {Ai intersect Aj}$\neq\emptyset$ contradicting the definition of Ti and Tj. Therefore, {t1,t2,...,tm} is a subset of Ti for only one i.

By the definition of power-setting, P(T) yields delta'(a,S)={S1,S2,...,Sm} and since {t1,t2,...,tm} is a subset of only one Ti, it follows that P(Ti) yields delta''(a,S)={S1,S2,...,Sm}, the unique transformation on a. Thus delta'=delta''.

The important part of this theorem is that while the order

of magnitude of powersetting T is O(c2**|T|) (where |T| denotes the number of elements of T), the order of magnitude of powersetting T1, T2, ..., Tn is:

O(c2**|T1|) + O(c2**|T2|) + ... + O(c2**|Tn|)

which is considerably smaller since |T| = |T1| + |T2| + ... + |Tn|. In the case of the start state of the NDFSM, this saving can be quite significant. For example, consider the start state of the NDFSM for the lexeme set of the language ALGOL-W given in Appendix B.

```
To state S1 on (start of the lexeme blankstring)
to state S2 on A-Z (start of the lexeme identifier)
to state S3 on C (start of the lexeme 'COMMENTcharseq;'
to state S4 on 0-9 (start of the lexeme integer)
to state S5 on # (start of the lexeme hexadecimal constant)
to state S6 on " (start of the lexeme string)
to state S7 on ; (start of the lexeme ';')
to state S8 on , (start of the lexeme ',')
to state S9 on : (start of the lexeme ':')
to state S10 on . (start of the lexeme '.')
To state S11 on ( (start of the lexeme '(')
to state S12 on ) (start of the lexeme ')')
to state S13 on + (start of the lexeme '+')
to state S14 on - (start of the lexeme '-')
to state S15 on * (start of the lexeme '*')
to state S16 on / (start of the lexeme '/')
to state S17 on * (start of the lexeme '**')
to state S18 on ¬ (start of the lexeme '¬')
to state S19 on | (start of the lexeme '|')
to state S20 on = (start of the lexeme '=')
to state S21 on ¬ (start of the lexeme '¬=')
to state S22 on < (start of the lexeme '<')
to state S23 on < (start of the lexeme '<=')
to state S24 on > (start of the lexeme '>')
to state S25 on > (start of the lexeme '>=')
to state S26 on : (start of the lexeme '::')
to state S27 on : (start of the lexeme ':=')
to state S28 on ' (start of the lexeme '''')
to state S29 on '01' (start of the lexeme end-of-file)
```

For this state, the set T is:

{S1,S2,...,S29}

which splits up into the subsets:

T1 = {S1}
T2 = {S2,S3}
T3 = {S4}
T4 = {S5}
T5 = {S6}
T6 = {S7}
T7 = {S8}
T8 = {S9,S26,S27}
T9 = {S10}
T10 = {S11}
T11 = {S12}
T12 = {S13}
T13 = {S14}
T14 = {S15,S17}
T15 = {S16}
T16 = {S18,S21}
T17 = {S19}
T18 = {S20}
T19 = {S22,S23}
T20 = {S24,S25}
T21 = {S28}
T22 = {S29}

Since there is no need to powerset any Ti which contains only one element, it follows that the execution time of the modified algorithm is:

TIME = O(C2**|T2|) + O(C2**|T8|) + O(C2**|T14|) +
O(c2**|T16|) + O(c2**|T19|) + O(c2**|T20|)
= O(4c) + O(8c) + O(4c) + O(4c) + O(4c) + 094c)
= O(28c)

which is much better than the execution time for the original

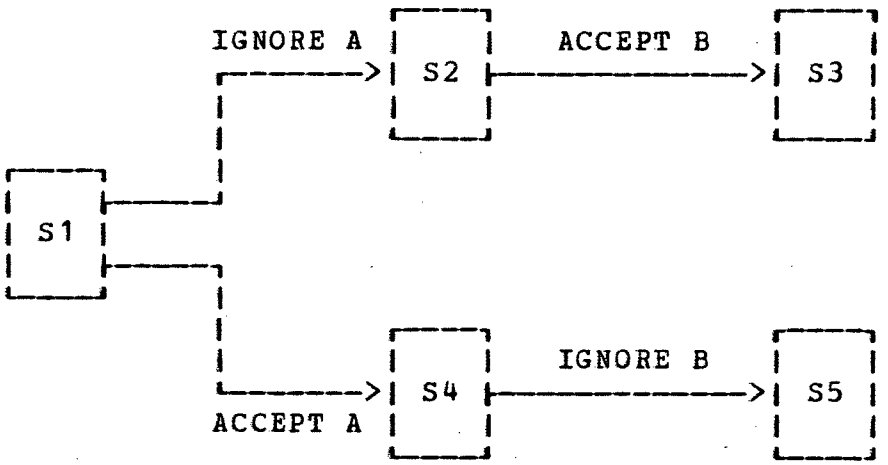
algorithm, $O(c2^{29})$ or O(c*500,000,000).

3.2.6 Checking the correctness of the DFSM

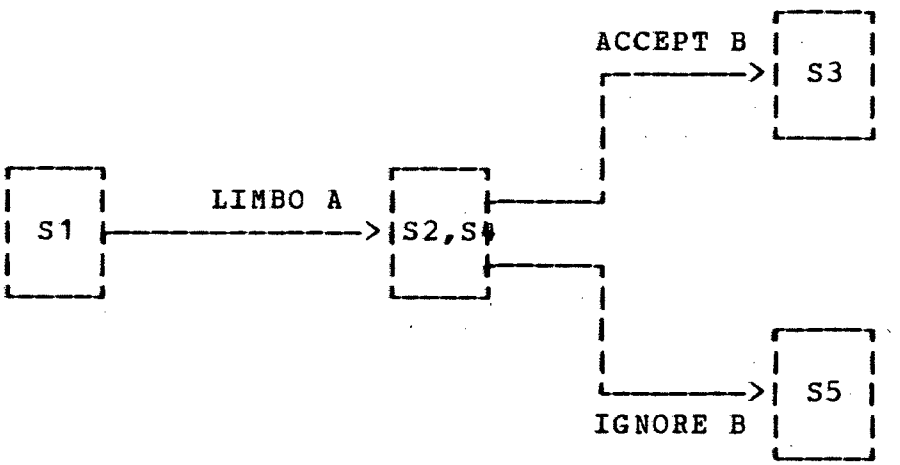
Although the modified algorithm for the construction of the DFSM does form limbo transitions, it still does not take into account the fact that creating the DFSM can cause transition sequences to occur which are not possible to emulate in the lexical scanner. The phase of the processing which checks this occurs immediately after the DFSM has been created.

The problem which occurs is that a limbo transition is effected in the final lexical scanner by having the lexical scanner issue a HOLD instruction to the lexical machine. As was previously mentioned, once a character is in the hold of the lexical machine, it must be removed before further processing of the input stream can occur. In terms of the DFSM, this means that a sequence of two or more limbo instructions cannot be allowed.

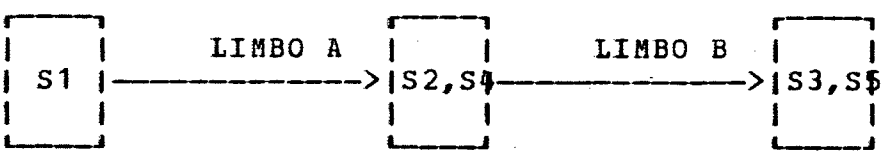
The phase which creates the DFSM does not check for this. If the above algorithm for the construction of the DFSM encounters a structure of the form:



it modifies it to be of the form:



and then to:



The final structure cannot be represented in the lexical machine since it requires two characters to be put into the hold. It should also be noted that there would be a lurking ambiguity if a structure of this form were to be allowed in the system since once state {s3,s5} was reached there would be no way to tell whether to ignore a and accept b or ignore b and accept a; the problem coming from the fact that if the five original states were labelled:

S1 to S2 ignore a

S2 to S3 accept b

S1 to S4 accept a

S4 to S5 ignore b

then the computation of the DFSM would yield the same final structure as above.

In addition to the analysis of the limbo transitions created during creation of the DFSM, this phase also checks for any ambiguities in the simple lexical machine action for a final state. Since the creation of the DFSM could join two states which are final and each final state in the system stands for the return of a particular lexeme, an ambiguity could arise in the lexeme which is to be returned from the combined state. As an example of this, consider the descriptions:

ONE OF "ABCDEFGHIHJKLMN", ANY OF "ABCDEFGHIJKLMN"

and

"BEGIN"

both of which will accept the word "BEGIN", a fact which is detected by checking for the final state ambiguity described above.

After this phase has been completed, it is then possible to punch out either a description language or PL360 source code representation of the lexical scanner.

3.2.7 Punching the Machine

This phase is included here for completeness rather than for its significance. The only point of interest is that as part of the punching phase, the transitions are sorted so that transitions from the state to itself ('WHILE' transitions) are punched first. Also an attempt is made to ensure, if possible, that if a transition exists on every character out of a state, then the last transition punched transfers control to the next state by "falling into it" in the sequential execution of instructions. In other words, a statement 'GO s2' is not punched for the last transition of a state s1 if it is possible to punch state s2 immediately after punching state s1.

The analysis done in this phase is independent of whether the intent is to punch the machine in the description language or in PL360 source code.

Chapter 4

4.1 Conclusion

The lexical scanner which is produced via the above set of transformations provides the language designer with one of the tools required in the construction of a compiler. The system allows for the generation of a very large class of lexical scanners and as such is useful in an equally large number of applications. Since the input language to the system is similar to the user's notion of what it is he is working with, the system is comfortable to work with for even the inexperienced user.

There are, of course, many other components required in order to build a compiler: reserved word recognizers, parsers, tree builders and code generators are just some of the components which could make up a compiler. Some of these components, such as parsers, have caused much interest and as such have been analyzed in detail. On the other hand, the problems involved in linking a parser generator into a compiler generator have received little attention.

Other components have been investigated indirectly; a generation program for the recognition of reserved words does not exist, but the theory of hashing functions which would be useful to the designer of such a generator does exist. Still other components have not been investigated at all, such as ambiguity scanners (for the solution of the open and close symbol problem in ALGOL-68). Finally, there are those components for which a formal methodology is still lacking,

these being in the aspects of the semantic interpretation of a language and how to include this into a compiler generation system.

Each of these components opens a door to an area in which further research is needed. Once some of these areas have been entered and generation systems developed for each of these components, then and only then will the truly modular compiler generation system have been achieved.

BIBLIOGRAPHY

[1] Feldman, J. A. A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler, Communications of the ACM, Volume 9, Number 1, 1966

[2] van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsay, C. H., Meertens, L. G. L. T., Fisker, R. G. (editors) Revised Report on the Algorithmic Language ALGOL-68, Springer-Verlag, New York, 1974

[3] Beautier, D. A. PSAP: A Guide for Users, University of California at Santa Barbara, 1973

[4] Johnson, W. L., Porter, J. H., Ackley, S. I., Ross, D. T. Automatic Generation of Efficient Lexical Processors Using Finite State Techniques, Communications of the ACM, Volume 11, Number 12, 1968

[5] Richards, M. BCPL: A Tool for Compiler Writing and System Programming, Spring Joint Computer Conference, AFIPS Press, 1969

[6] PL360 Programming Manual, University Printing Service, Newcastle, 1972

[7] Warshall, S. A Theorem on Boolean Matrices, Communications of the ACM, Volume 9, Numer 1, 1962

[8] Warren, H. S. Jr. A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations, Communications of the ACM, Volume 18, Number 4, 1975

[9] ALGOL-W Programming Manual, University Printing Service, Newcastle, 1970

[10] Aho, A., Hopcroft, J. E., Ullman, J. D. The Design and Analysis of Computer Algorithms, Addison-Wesley, Don Mills, 1974

Appendix A

An Input Description for ALGOL-W

The following is an example of the type of input which is accepted by the lexical scanner generation system. This example is for the language ALGOL-W.

```
BEGIN
   LEXEME 1 IS " ", ANY OF " ".
   LEXEME 2 IS ONE OF "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
   ANY OF "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_".
   LEXEME 3 IS "COMMENT", NOTANY OF ";", ";".
   LEXEME 4 IS ONE OF "0123456789", ANY OF "0123456789".
   LEXEME 5 IS "#", ONE OF "0123456789", ANY OF "0123456789".
      SUBCHAR IS NOTANY OF """" OR IGNORE """", """".
   LEXEME 6 IS IGNORE """", ANY OF SUBCHAR, IGNORE """".
   LEXEME 7 IS ";".
   LEXEME 8 IS ",".
   LEXEME 9 IS ":".
   LEXEME 10 IS ".".
   LEXEME 11 IS "(".
   LEXEME 12 IS ")".
   LEXEME 13 IS "+".
   LEXEME 14 IS "-".
   LEXEME 15 IS "*".
   LEXEME 16 IS "/".
   LEXEME 17 IS "**".
   LEXEME 18 IS "¬".
   LEXEME 19 IS "|".
   LEXEME 20 IS "=".
   LEXEME 21 IS "¬=".
   LEXEME 22 IS "<".
   LEXEME 23 IS "<=".
   LEXEME 24 IS ">".
   LEXEME 25 IS ">=".
   LEXEME 26 IS "::".
   LEXEME 27 IS ":=".
   LEXEME 28 IS "''".
   LEXEME 28 IS "'1'".
END
```

The following is the lexical scanner (given in the description language) for the ALGOL-W lexeme description given

above.

```
S95 IF " " (ACCEPT GO S2)
    IF "C" (ACCEPT GO S97)
    IF "ABDEFGHIJKLMNOPQRSTUVWXYZ" (ACCEPT GO S5)
    IF "0123456789" (ACCEPT GO S19)
    IF "#" (ACCEPT GO S22)
    IF """ (IGNORE GO S33)
    IF ";" (ACCEPT RETURN 7)
    IF "," (ACCEPT RETURN 8)
    IF "." (ACCEPT RETURN 10)
    IF "(" (ACCEPT RETURN 11)
    IF ")" (ACCEPT RETURN 12)
    IF "+" (ACCEPT RETURN 13)
    IF "-" (ACCEPT RETURN 14)
    IF "/" (ACCEPT RETURN 16)
    IF "*" (ACCEPT GO S98)
    IF "|" (ACCEPT RETURN 19)
    IF "=" (ACCEPT RETURN 20)
    IF "¬" (ACCEPT GO S99)
    IF "<" (ACCEPT GO S50)
    IF ">" (ACCEPT GO S51)
    IF ":" (ACCEPT GO S52)
    IF "'" (ACCEPT RETURN 28)
    IF "?" (ACCEPT RETURN 29) ELSE ERROR
S52 IF ":" (ACCEPT RETURN 26)
    IF "=" (ACCEPT RETURN 27)
    (RETURN 9)
S51 IF "=" (ACCEPT RETURN 25)
    (RETURN 24)
S50 IF "=" (ACCEPT RETURN 23)
    (RETURN 22)
S99 IF "=" (ACCEPT RETURN 21)
    (RETURN 18)
S98 IF "*" (ACCEPT RETURN 17)
    (RETURN 15)
S33 IFNOT """ (ACCEPT GO S36)
    (IGNORE )
S53 IF """ (ACCEPT GO S40)
    (RETURN 6)
S40 IFNOT """ (ACCEPT GO S36)
    (IGNORE GO S53)
S36 WHILENOT """ (ACCEPT )
    (IGNORE GO S53)
S22 IF "0123456789" (ACCEPT ) ELSE ERROR
S24 IF "0123456789" (ACCEPT GO S25)
    (RETURN 5)
S25 WHILE "0123456789" (ACCEPT )
    (RETURN 5)
S19 IF "0123456789" (ACCEPT GO S20)
    (RETURN 4)
S20 WHILE "0123456789" (ACCEPT )
```

```
    (RETURN 4)
S5  IF "_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" (ACCEPT GO S6)
    (RETURN 2)
S6  WHILE "_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" (ACCEPT )
    (RETURN 2)
S97 IF "O" (ACCEPT GO S54)
    IF "_ABCDEFGHIJKLMNPQRSTUVWXYZ0123456789" (ACCEPT GO S6)
    (RETURN 2)
S54 IF "M" (ACCEPT GO S55)
    IF "_ABCDEFGHIJKLNOPQRSTUVWXYZ0123456789" (ACCEPT GO S6)
    (RETURN 2)
S55 IF "M" (ACCEPT GO S56)
    IF "_ABCDEFGHIJKLNOPQRSTUVWXYZ0123456789" (ACCEPT GO S6)
    (RETURN 2)
S56 IF "E" (ACCEPT GO S57)
    IF "_ABCDFGHIJKLMNOPQRSTUVWXYZ0123456789" (ACCEPT GO S6)
    (RETURN 2)
S57 IF "N" (ACCEPT GO S58)
    IF "_ABCDEFGHIJKLMOPQRSTUVWXYZ0123456789" (ACCEPT GO S6)
    (RETURN 2)
S58 IF "T" (ACCEPT GO S59)
    IF "_ABCDEFGHIJKLMNOPQRSUVWXYZ0123456789" (ACCEPT GO S6)
    (RETURN 2)
S59 IF "_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" (ACCEPT GO S60)
    IFNOT ";_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
                    (ACCEPT GO S15)
    (ACCEPT RETURN 3)
    (RETURN 2)
S15 WHILENOT ";" (ACCEPT )
    (ACCEPT RETURN 3)
S60 WHILE "_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" (ACCEPT )
    IFNOT ";_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
                    (ACCEPT GO S15)
    (ACCEPT RETURN 3)
    (RETURN 2)
S2  IF " " (ACCEPT GO S3)
    (RETURN 1)
S3  WHILE " " (ACCEPT )
    (RETURN 1)
```

Appendix B
Using the
Lexical Scanner Generator

The following is a description of how to use the lexical scanner generator as it is implemented at UBC.

The purpose of the lexical scanner generation system is to supply the compiler writer with an easy method for generating lexical scanners for various programming or command languages.

Availability

The system can be invoked via the following MTS run command:

```
$RUN TEDV:LEXGENERATOR SCARDS=inputfile SPRINT=listingfile
        SPUNCH=outputfile {PAR=parfield}
```

where each of the values expressed in lower case letters is defined as follows:

inputfile - This is the file which contains the description of the lexemes in accordance with the syntax of the Lexical Scanner Generation System.

listingfile - This is the file onto which the Lexical Scanner Generation System will both echo the

input lines and print any information which was requested via the PAR= field of the $RUN command.

outputfile - This is the file onto which the lexical Scanner Generation System will punch either the ideal machine or the PL360 coded version of the ideal machine.

parfield - The value of 'parfield' determines whether to produce ideal machine code or PL360 code and also determines whether any tracing information is to be produced. The 'parfield' is made up of a sequence of items separated by commas and is evaluated from left to right. An item can be any of the following:

TRACE(charseq) - The 'charseq' is scanned from left to right and any digit n (0 <= n <= 6) which is found causes trace flag n to be turned on. The actions of the various flags are:

0 - All trace flags are turned off.

1 - All trace flags are turned on.

2 - The initial machine before any transitions have been performed is printed out.

3 - The empty transition matrix is printed out.

4 - The transitive closure of the empty transition matrix is printed out.

5 - The non-deterministic machine is printed out.

6 - The deterministic machine is printed out.

PL360 - Instead of punching out the ideal machine, the Lexical Scanner Generation System will punch out a PL360 source program which can then be compiled giving a Fortran callable module.

If the lexical scanner generation system is run with PL360 in the PAR= field, then the resulting PL360 source program can be compiled using the following MTS $run command:

```
$RUN ALGW:PL360M SCARDS=outputfile SPRINT=listingfile
                 SPUNCH=deckfile
```

where each of the values expressed in lower case letters is defined as follows:

outputfile - This is the outputfile as specified from the SPUNCH=outputfile on the $RUN command of the Lexical Scanner Generation system.

listingfile - This is the file on which the PL360 compiler will produce a compilation listing.

deckfile - This is the file onto which the PL360 compiler will punch the object deck.

The object deck which results from this compilation is a Fortran callable module called GETLEX with the following calling sequence:

```
INTEGER LEXNUM,LENGTH
LOGICAL*1 LEXEME(n)
      .
      .
      .
CALL GETLEX(LEXNUM,LENGTH,LEXEME)
      .
      .
      .
```

The GETLEX module sets the value of LEXNUM to the integral

value associated with the lexeme, LENGTH to the length of the symbolic string which composes the lexeme and LEXEME to the symbolic represetation of the token. The user must use a value of 'n' in the declaration of LEXEME such that n is greater than or equal to the maximum length of any LEXEME which could be returned.

The module also requires that the user supply two Fortran callable modules with the following names and calling sequences:

```
      SUBROUTINE NEWLIN(LENGTH,BUFFER)
      INTEGER LENGTH
      LOGICAL*1 BUFFER(256)
C     SET THE VALUE OF BUFFER TO THE NEXT
C     INPUT BUFFER AND LENGTH TO ITS LENGTH
            .
            .
            .
      END


      SUBROUTINE LEXERR(CHAR)
      INTEGER CHAR
C     CHAR HAS CAUSED AN ERROR IN
C     GETLEX.  IF UNCHANGED THE CHAR
C     IN ERROR WILL BE DELETED,
C     OTHERWISE THE OLD VALUE WILL
C     BE REPLACED BY THE NEW VALUE
            .
            .
            .
      END
```