A PORTABLE LEXICAL ANALYZER WRITING SYSTEM

RAND Corporation

PREPARED FOR
National Science Foundation

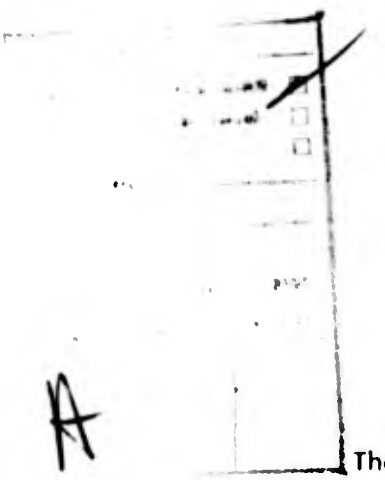
November 1973

A PORTABLE LEXICAL ANALYZER WRITING SYSTEM

Robert C. Gammill

November 1973

P-5117



The Rand Paper Series

The Rand Corporation
Santa Monica, California 90406

Abstract

A PORTABLE LEXICAL ANALYZER WRITING SYSTEM

LAWS, a lexical analyzer writing system implemented in FORTRAN, attempts to achieve a compromise between the contradictory demands of efficiency and portability. Simplicity is emphasized to make extension and modification easy. LAWS is a compiler for generating state transition tables and a finite state machine (FSM) directed by those tables. The language (STATE-DEF) used to program the FSM resembles Floyd-Evans Production Language [1,2]. A program written in STATE-DEF, the resulting state transition table, and a flow chart of the FSM are given. Of special interest are the methods for keeping tables small and automatically translating character codes.

1. INTRODUCTION

Lexical analyzers are used in the front end of many kinds of language processors, especially compilers. The analyzer (sometimes called a scanner) collects a stream of input characters into symbols for use by the rest of the processor. Examples of symbols might include variable names, constants and separators. Thus, the following FORTRAN card (1) could be transformed by a scanner into the stream of symbols (2).

```
IF(1.E5.GT.EQ5)GOTO70=4.0                    (1)
IF ( 1.E5 .GT. EQ5 ) GOTO70 = 4.0            (2)
```

Since the analyzer must handle every input character, it is important that it be efficient. Since the specification of a lexical analyzer is primarily language dependent, it is desirable for a single specification to work on many different machines. However, efficiency and portability tend to be contradictory demands. A portable lexical analyzer writing system (LAWS) that attempts to achieve a reasonable compromise has been designed and implemented. LAWS is implemented in FORTRAN. It is composed of a compiler and an interpreter, as shown in Fig. 1. The compiler accepts a program written in a language called STATE-DEF [5] and generates a state transition table. That state transition table and the interpreter form a finite state machine (FSM) whose transitions have associated action code numbers. When the FSM is combined with a set of semantics routines, (one per action code number), the complete lexical analyzer has been defined.

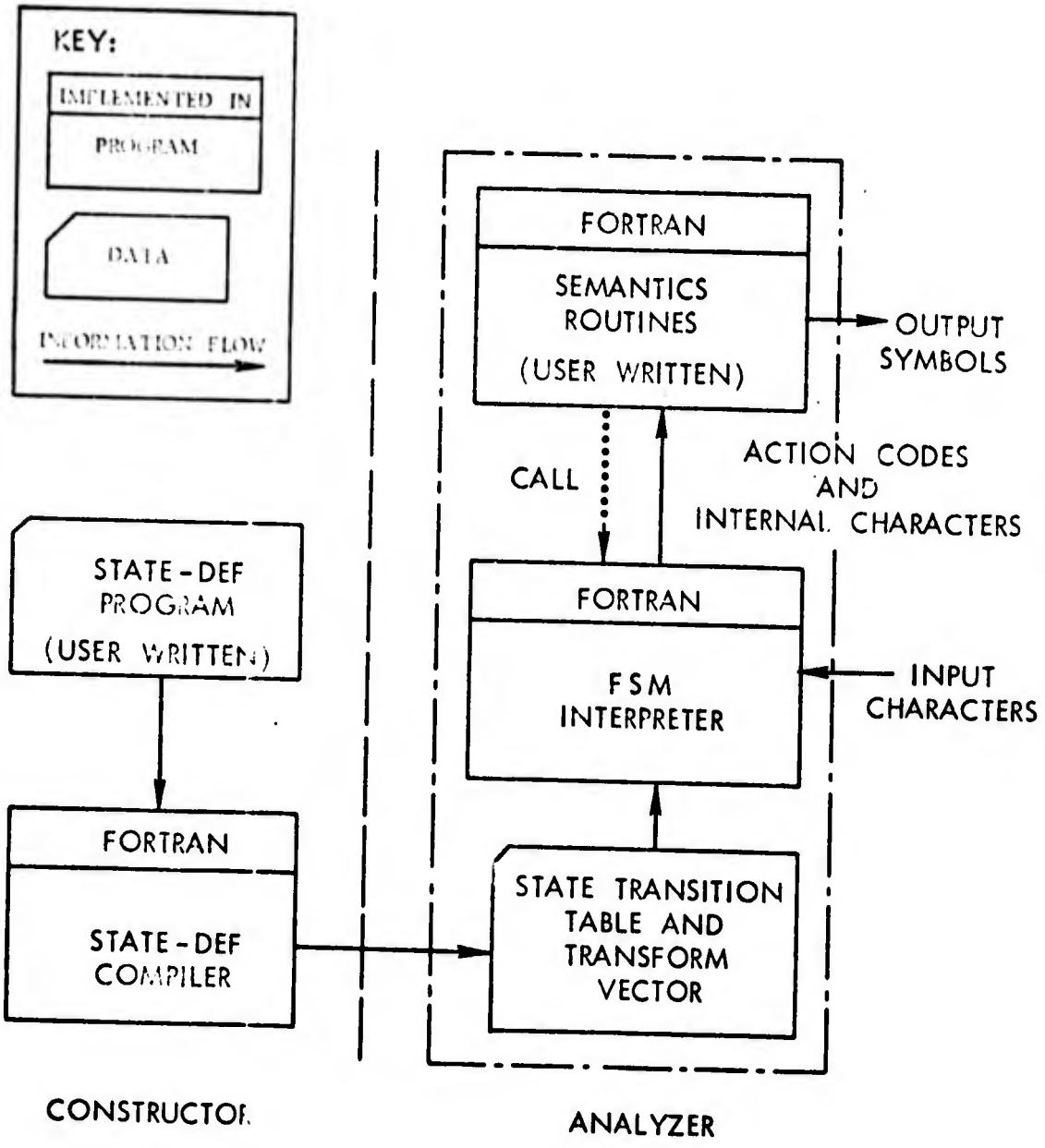


Fig. 1 — Flow of information between elements of LAWS

2. EARLIER WORK

Gries [3] gives a good discussion of the issues involved in the use and construction of lexical analyzers. Two general methods of constructing lexical analyzers have been used in the past. The earliest approach, still used by many compiler implementors, is to write the analyzer in a general purpose programming language such as ALGOL or assembly language. This approach often produces an analyzer which has many conditional (or case) statements and a complex structure. The result can be difficult to modify and is sometimes inefficient as well.

More recently work has proceeded on scanner construction systems. These systems provide a language, and other mechanisms (like those shown in Fig. 1), especially suited for the easy specification of efficient lexical analyzers. The best known of these is the AED RWORD System [4].

The RWORD system uses a declarative language based on regular expressions and produces an optimal lexical analyzer using techniques from the theory of finite automata. RWORD appears to be a powerful and effective tool for the construction of lexical analyzers. However, it has certain drawbacks. Among these are the size and complexity of the elements which make up the analyzer constructor. RWORD has two phases, the first of which constructs an AED-0 language version of the FSM, and the second of which constructs a macro-assembly language version. Inherent in this process is the need for an AED-0 compiler and a macro-assembler. Furthermore, most of the elements are constructed

in the AED language, which is not universally available. Thus, although RWORD is clearly a powerful tool, its complexity and limited portability have kept it from wide use.

LAWS is a considerably less powerful tool, requiring the user to program a state transition table in STATE-DEF. That language resembles Floyd-Evans Production language [1,2], and shares its simplicity. The compiler and interpreter are implemented in FORTRAN, and are relatively compact and simple programs. Portability and efficiency have been emphasized. As a result, the comparison between LAWS and RWORD is analagous to that between a bicycle and a luxurious automobile. Where RWORD is complex and powerful, LAWS is simple and direct. It is hoped that the virtues of clarity, simplicity and portability will serve to overcome the limitations.

3. WRITING LEXICAL ANALYZERS

STATE-DEF (State Table Definition) was created by Schwanke [5] as a convenient and easily understood tool for defining state tables. His inspiration came from the clarity and simplicity evident in the Floyd-Evans Production Language used for defining syntax analyzers. STATE-DEF is used for defining finite state languages and is itself a finite state language (STATE-DEF has been used to define itself). Like Production Language, STATE-DEF allows semantic code numbers (action codes) to be associated with its transitions. Finite state machines tend to be quite efficient, and despite the implementation of the interpreter in FORTRAN

(inherently inefficient for this kind of task) satisfactory processing speeds have been achieved. The simplicity and small size of the interpreter assure its easy conversion to optimized machine language for operational use.

STATE-DEF is used to define state tables. A state table is a two dimensional array with rows representing states and columns representing input characters. The entry at any table position specifies the next state (a transition) and an action code (semantic number). An example of a state table and its STATE-DEF program are shown in Fig. 2. The period is used as a flag character in STATE-DEF to indicate that the specification of an input character follows.

Input Character

State Name	X	Y
A	A1	B2
B	B2	A3

```
State      Input       Next          Action
Name       Character   State         Code
A:           .X        NEXT=A,       ACTION=1;
             .Y        NEXT=B,       ACTION=2;
B:           .X        NEXT=B,       ACTION=2;
             .Y        NEXT=A,       ACTION=3;
END:          INITIAL STATE=A;
```

Fig. 2. A State Table and its STATE-DEF Program.

One problem with state tables is that if the input alphabet is large, and one column is allowed for each character, then the state table will be very wide. However, numerous input characters will cause exactly the same transition and action. These characters may be thought of as forming a class. If we translate the input character into a class number before indexing into the state table, we can save space due to the elimination of redundant transitions. For example, all the alphabetic characters might receive the class number 1. To transform characters into class numbers, we use a vector with the character codes as indices as shown in Fig. 3.

INPUT CHARACTER	CLASS NUMBER	MEANING
A	1	ALPHABETIC
Z	1	
0	2	NUMERIC
9	2	
+	3	SPECIAL
/	3	
	0	IGNORE

Fig. 3. A Vector for Converting Characters to Class Numbers

One difficulty is that in some states we need to separate alphabetic and numeric characters, while in another state we may need to separate the first letters of key words

from other alphabetic characters. Clearly, it would be helpful to have a different transform vector for every state. This would be uneconomical if we stored each transform vector in a separate array. However, we note that under normal circumstances the number of different classes per state will be small. So, although a transform vector will be 64 or 256 elements long, it will be only a few bits wide (2 bits in the case of the vector shown in Fig. 3, and a maximum of 6 bits for a 64 character input set).

The obvious solution is to pack the transform vectors for different states into a single overall transform vector. In each state we will have to remember which bits of the overall transform vector represent the class numbers for the input characters in this state. This means that the information needed to define a state will now have three components (STATE, INSET, MASK), as illustrated in Fig. 4. If we are in state (IDENT, 10, 2) and the input character is a %, the packed transform vector will be right shifted by 10 bits and the rightmost 2 bits selected to produce a class code of 2.

Another use of the transform vector is to include an internal form of the character code, either to allow a special internal collating sequence or change from, say, EBCDIC to BCD. If all of this information can be packed into a space which can be accessed by a single storage reference, we are able to achieve considerable efficiency. If that is not possible initially, it should be clear that some merging of the information can be achieved which will make it possible. The method for compacting

	Class Numbers			Internal Character Code
A	2	1	1	A
Z	2	1	1	Z
<	2	2	3	)
0	1	1	2	0
%	2	2	3	(
9	1	1	2	9
≡	0	0	0	zero
=	2	2	3	=
(	2	2	3	(
	0	0	0	zero
	2	2	2	8 BITS

Table for Finding Class No.

STATE	INSET	MASK
START	8	2
IDENT	10	2
NUMBER	12	2

INSET = number of bits to right shift

MASK = number of useful bits after shift.

Fig. 4

Three Transform Vectors and
Internal Character Code Packed Together

the overall transform vector has been examined and is uncomplicated. It has not yet been implemented.

To summarize, when an external input character code is collected it is used to index the transform vector. This provides a bit string, assumed to be less than one word in length, which is called the internal character code. The right hand end of the bit string is the actual internal coded character, suitable for printing. The remainder of the bit string is a sequence of packed class codes. The state information of the FSM has been extended to include information called INSET and MASK which specifies the number of bits to be shifted and masked to retrieve the class code number for the character in this state. If the class code turns out to be zero, then the character may be discarded immediately. Otherwise, the class code is added to the state value to find the location of a transition in the state table. Because of the class numbers, it is no longer necessary for the state table to be rectangular. Instead the state becomes a base value in a linear vector of transitions. The actual transition is selected (with respect to the base value) by the class code. An interesting feature of this arrangement is that if the same character is used to produce more than one transition (a common technique in lexical analysis) a new class code can be retrieved without any more table references.

Fig. 5 shows a STATE-DEF program and the state table generated. Two new features have been introduced in the STATE-DEF program. These are the DEF card, which allows a collating sequence to be specified, and the character class specification (e.g.,.A-Z), which

```
DEF  46 ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789=+-*/(),.$

START:   .A-Z  ACTION=1, NEXT=IDENT;     0
         .0-9  ACTION=1, NEXT=NUMBER;    1
         .=-$  ACTION=2;                 2
IDENT:   .A-9  ACTION=1;                 3
         .=-$  ACTION=3, NEXT=START;     4
NUMBER:  .0-9  ACTION=1;                 5
     .A-Z.=-$  ACTION=4, NEXT=START;     6

END:           INITIAL STATE=START;
```

	STATE	INSET	MASK	ACTION
0	3	10	2	1
1	5	12	2	1
2	0	0	0	2
3	0	0	0	1
4	0	8	2	3
5	0	0	0	1
6	0	8	2	4
	3	4	2	3 BITS

Fig. 5

A STATE-DEF Program for a Three State FSM and the State Table Generated by the Compiler

uses the specified collating sequence to pick out a group of characters (A through Z). Note that the character transform vector shown in Fig. 4 goes with the state table of Fig. 5.

4. THE INTERPRETER

A flow chart of the interpreter for LAWS state tables is given in Fig. 6. This flow chart is extremely simple to implement. The most important aspect of the implementation is handling the problems of shifting and masking in a portable manner. This has been accomplished by providing a set of primitive functions written in machine language for use by the FORTRAN program (LEFTSH, RGHTSH, OR and AND). Such functions exist on most binary computers. When they do not exist, they can be easily implemented. It should be noted, however, that decimal machines (without binary shifting capabilities) or very short word length machines (e.g. some mini-computers) will probably not be suitable for execution of a LAWS lexical analyzer.

No flow chart has been given for the program containing the semantics routines mentioned in Fig. 1. That program usually has such a simple structure as not to merit a flow chart. Normally it is a subprogram which is called every time the language processor needs a new symbol (token). The subprogram first calls the interpreter (shown in Fig. 6). Return from the interpreter occurs when an action code has been found, so a computed GO-TO can be executed to one of a set of semantic routines. After completion of the semantic routine, if a complete symbol is ready, control returns

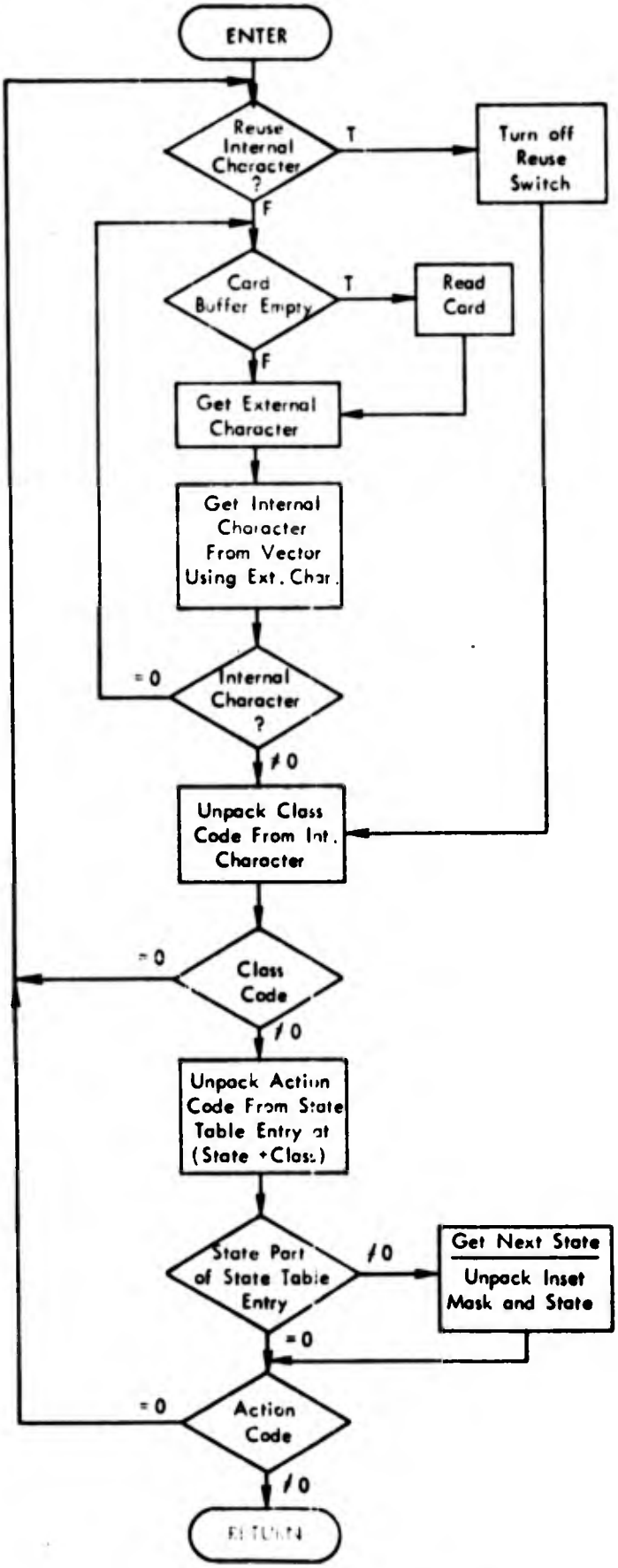


Fig. 6--Flow chart of the interpreter

to the caller. If not, control loops back and another call is made on the interpreter. In the case of the STATE-DEF program of Fig. 5, four action codes were specified. The required semantics for each of those codes is listed in the table below.

TABLE OF ACTIONS

ACTION NUMBER	MEANING
1	QUEUE THE INTERNAL CHARACTER
2	RETURN THE INTERNAL CHARACTER AS A SYMBOL (SPECIAL CHAR)
3	RETURN THE QUEUE OF CHARACTERS (IDENTIFIER). SET REUSE SWITCH.
4	RETURN THE QUEUE OF CHARACTERS (NUMBER). SET REUSE SWITCH.

5. SUMMARY

A portable lexical analyzer writing system (LAWS) has been implemented in FORTRAN. It is an extremely simple system, yet is powerful despite that simplicity. Special care is used in LAWS to achieve compactness of the state table and character translation vector. The simplicity of the interpreter which runs from those tables makes conversion to machine language easy. Implementation in FORTRAN and careful use of primitive shifting and masking functions maximizes portability and enhances possibilities for tailoring to user needs.

6. ACKNOWLEDGMENT

This research was supported by the National Science Foundation, Office of Computing Activities, under grant GJ-36417.

7. REFERENCES

[1] Arthur Evans Jr., An ALGOL 60 Compiler, Annual Review in Automatic Programming, 1964, 87-124.

[2] R. W. Floyd, A Descriptive Language for Symbol Manipulation, Journal of the ACM, vol. 8, no. 4, October 1961, 579-584.

[3] David Gries, Compiler Construction for Digital Computers, John Wiley & Sons, New York, 1971.

[4] Walter L. Johnson, James H. Porter, Stephanie I. Ackley and Douglas T. Ross, Automatic Generation of Efficient Lexical Processors Using Finite State Techniques, Communications of the ACM, vol. 11, no. 12, December 1968, 805-813.

[5] Lee M. Schwanke, MACS - A Programmable Pre-Processor With Macrogeneration Facilities, Masters Thesis, University of Colorado, 1972.