Presented at the USENIX Association and Software Tools User Group Summer Conference, Salt Lake City, UT, June 12-15, 1984

A PORTABLE, PUBLIC DOMAIN LEX FOR THE SOFTWARE TOOLS

V. Paxson

June 1984

TWO-WEEK LOAN COPY

This is a Library Circulating Copy which may be borrowed for two weeks.

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098
DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.
A Portable, Public Domain LEX for the Software Tools

Vern Paxson

Real Time Systems Group
Lawrence Berkeley Laboratory
University of California
Berkeley, California, 94720

1. Introduction

Chances are you're writing a lot of your programs the hard way. You can write smaller, more efficient programs with less effort by using the portable Software Tools Lex. This paper describes how to use Lex and why programs generated with it give excellent performance.

At the lowest level, programs have to deal with their input on a character by character basis. Usually what's interesting about the input is not the individual characters but the patterns they come in. For example, a string of digits is only meaningful when the digits are grouped together as a number. A group of characters that should be considered as a unit is often called a token, and recognizing tokens is called scanning. Almost any program whose input has some structure will use a scanner to recognize that structure.

Because a scanner often has to recognize a large number of different, complicated patterns, you can spend a lot of time trying to code an efficient scanner by hand. There are techniques for automatically constructing highly efficient scanners given a description of the patterns the scanner should recognize. Using these techniques, Lex makes it easy for you to create scanners that are more efficient than those you might code by hand.

2. Using Lex

The Lex user-interface is based on the user-interface of the UNIX¹ Lex program, which is simple but powerful. The user specifies a scanner by a set of rules. Each rule is made up of a pattern, which is written using an extended set of regular expressions, and an associated action, which is a sequence of Ratfor code. Lex's output is a Ratfor function which is compiled and linked into the user's program. When executed, the generated function reads characters from the input. Whenever one of the patterns is matched by a string of input characters, the action associated with the pattern is executed.

For example, a rule specifying that whenever the word "FOOBAR" is seen a counter should be incremented would look like:

```
FOOBAR { count = count + 1 }
```

The extended set of regular expressions allows complicated patterns to be specified. Figure 1 shows a Lex rule which recognizes Ratfor function declarations and returns an appropriate token to the caller of the scanning routine. The "\" (alternation) operator specifies that any one of the words "integer", "character", "logical", or "real" may be matched. The parentheses group these types together so that the next operator, the "\" character, operates on them as a whole. "\" specifies that the preceding subpattern is optional. This is necessary because it is legal in Ratfor to declare a function without specifying its type. The brackets ("[" and "]") that follow indicate a character class that matches either a blank or a tab. The "*" operator is positive closure. It means "match one or more times". Thus "[0-9]+" matches any series of one or more blanks or tabs.

The rule will match and perform its action on any of the following inputs:

```
real function
integer function
```

Fig. 1: A Lex rule to recognize Ratfor function declarations

The UNIX Lex program also provides both a macro facility to ease the construction of large rules and two mechanisms by which rules can be selectively turned on or off depending on textual context. Software Tools Lex supports all of the features of UNIX Lex (described in Lesk & Schmidt).
3. Generating Efficient Scanners

While regular expressions provide a natural way for the user to specify a scanner, it is difficult to write an efficient scanner using the natural pattern-matching techniques. Given a string of text, a natural way to see if it matches any one of several patterns is to attempt to match it against each pattern in turn. This method will, at best, take time proportional to the number of characters in the input times the number of characters in the patterns. There are often fifty or more patterns that need to be checked. Matching just one pattern can require several tests to see if all of the parts of the pattern are matched. For example, to test if the pattern shown in Figure 1 is matched a scanner must test the first part of the input string against “integer”, “character”, “logical”, and “real”. Thus, a scanner which uses the natural pattern-matching techniques has to do a lot of comparisons to recognize one string of text. Since the scanner processes every string in the input, the program is going to spend a great deal of its time doing these comparisons. The natural pattern-matching techniques lead to slow programs.

Surprisingly, there are pattern-matching techniques which will match an arbitrary number of patterns, each of arbitrary complexity, in a time that depends solely on the number of characters in the input string. These techniques are based on Finite Automaton (FA) theory. In spite of their power, they turn out to be quite simple. An FA simply consists of a current state and a transition table. The FA begins reading input in some initial state. Given the current state and the next input character, the transition table gives the next state the machine should enter. If the next input character does not match a pattern the FA jams, stops pattern-matching, and executes the action associated with the longest pattern matched so far.

The code to run a FA is shown in Figure 2. (The array nxtstate is the transition table.) Figure 3 shows a pictorial representation of an FA which recognizes the string “HELLO”. The numbered nodes in the picture represent states. The initial state is #1, and the final, accepting state is #6. If the accepting state is entered then the pattern has been matched. The arrows indicate state transitions. The labels on the arrows indicate which input symbols trigger those transitions. When executing this automaton using the code shown in Figure 2, nxtstate(3, ‘1’) is 4 and nxtstate(3, ‘o’) is JAM.

Matching multiple patterns simultaneously or matching patterns which contain closures or alternations requires something more complicated than the FA in Figure 3. To match either “HELLO” or “HI”, you can construct an FA like the one shown in Figure 4. This is just two simple matchers, one for “HELLO” and one for “HI”, connected by a new transition, e, which you could read as “maybe”. The idea is that, conceptually, you have two occurrences of the loop in Figure 2 running simultaneously. When the pattern-matcher starts it might in state #2 waiting for an “H” or it might be in state #8 waiting for an “H”. If an “H” is read then the pattern-matcher might be in state #3 waiting for an “E” or it might be in state #9 waiting for an “I”. It keeps running until all of the FAs have jammed. The last one to jam determines which pattern was matched.

This type of FA is called a Non-deterministic Finite Automaton (NFA) because it can be in more than one state at any given time. There exist straightforward algorithms for converting regular expressions into NFAs. Figure 5 shows an NFA which recognizes zero or more occurrences of the string “HI”. Such a pattern would be specified as “(HI)**” to Lex. Unfortunately, NFAs cannot be run using the simple pattern-matching code pictured in Figure 2, unless one has a multi-processor computer with a potentially infinite number of processors.

While it is not at all obvious, any NFA can be

---

Fig. 2: An FA to recognize “HELLO” and “HI”
integer nxtstate(MAXSTATES, MAXCHARS), action(MAXSTATES)

# Data to initialize nxtstate and action go here.

curstate = initstate

repeat

{ 
ch = getc(ch)
prevstate = curstate
curstate = nxtstate(curstate, ch)
}

until (curstate == JAM)

# Execute the action associated with the pattern matched.

switch(action(prevstate))

{

case RULE1:

    # code for the first rule goes here.

    

}

Fig. 2: Pattern-Matching Loop for a Finite Automaton
converted into an equivalent Deterministic Finite Automaton (DFA) which will recognize the same set of patterns as the NFA but can be run on a single processor using the fast code shown in Figure 2. The algorithm to do the conversion is called subset construction. The fundamental idea behind subset construction is that each state of the DFA represents a set of NFA states. For example, Figure 6 shows the DFA equivalent of the NFA shown in Figure 4. The DFA states are numbered to suggest their set structure. The initial state of the DFA, state #128, is equivalent to states #1, #2, and #8 of the NFA in Figure 4. Upon reading an “H”, the DFA enters state #39, which is equivalent to the set of NFA states #3 and #9. state #4 of the DFA is equivalent to the set consisting of the singleton NFA state #4, and so on. In this manner, when the DFA is in a given state, it is identical to the NFA being in each of the NFA states which are members of that DFA state. So each DFA state corresponds to a set of NFA states being active at the same time.

Since subset construction involves constructing and comparing a great number of sets, implementing it efficiently is a lot of work. Indeed, it took us two complete re-designs to get Lex to perform reasonably well. Although the experience was painful, it might be instructive and we plan to discuss the details in a future paper.


### 4. Generating Compact Scanners

While the pattern-matching algorithm shown in Figure 2 is simple and fast, it has one drawback: the two-dimensional array nxtstate can be very large. Table 1 shows the size of nxtstate for a variety of Lex inputs. Programs with tables this big won’t fit on some machines that run the Software Tools. Even on virtual memory machines these tables would consume a lot of run-time resources.

Fortunately it is possible to eliminate a great deal of redundancy in the transition table and use much less memory than the two-dimensional array requires.

To illustrate the source of the redundancy, we constructed a small scanner that has many of the features found in more complicated sets of rules. Consider the Lex input shown in Figure 7. These rules specify a scanner which will match "HELLO", "HI", and any string of letters. The first two patterns are keywords. The last pattern is a catch-all. It will match any string of letters which is not a keyword. Rules like these (several keyword rules and a few catch-all rules) are typical of most complicated scanners, including the five shown in Table 1.

![Diagram](image_url)

**Fig. 5: An NFA which recognizes "(HI)*"**

![Diagram](image_url)

**Fig. 6: A DFA to recognize "HELLO" and "HI"**

**Table 1: Sizes of some typical DFA transition tables**

<table>
<thead>
<tr>
<th>Input</th>
<th># of Rules</th>
<th># of Chars</th>
<th># of States</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>awk</td>
<td>89</td>
<td>128</td>
<td>202</td>
<td>25856</td>
</tr>
<tr>
<td>nwords</td>
<td>65</td>
<td>128</td>
<td>254</td>
<td>32512</td>
</tr>
<tr>
<td>ratfix</td>
<td>68</td>
<td>128</td>
<td>430</td>
<td>55040</td>
</tr>
<tr>
<td>sh</td>
<td>30</td>
<td>128</td>
<td>215</td>
<td>27520</td>
</tr>
<tr>
<td>typeof</td>
<td>13</td>
<td>128</td>
<td>215</td>
<td>27520</td>
</tr>
</tbody>
</table>

"awk" is the scanner for the UNIX Awk tool. "nwords" scans English text. "ratfix" converts old-style Ratfor programs to the standard Software Tools Ratfor. "sh" is a scanner for the Software Tools Sh program. "typeof" determines the type of text files (e.g., Ratfor source, Roff input, local operating system command file, etc.).
Fig. 8: An NFA which recognizes keywords and a catch-all

Transitions marked as "*" mean that the transition is made on any letter except the one which is explicitly shown as making another transition. For example, state #4 makes a transition to state #8 on any letter except "L". State #3 makes a transition to state #8 on any letter.

Fig. 9: The DFA equivalent
HELLO { return(HELLO_TOKEN) }
HI { return(HI_TOKEN) }
[A-Z]+ { return(IDENTIFIER_TOKEN) }

Fig. 7: A set of Keyword and Catch-all Lex rules

Table 2: Transition Table for the DFA

<table>
<thead>
<tr>
<th>Character</th>
<th>e</th>
<th>h</th>
<th>i</th>
<th>l</th>
<th>o</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>state 1</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>state 2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>state 3</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>state 4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>state 5</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>state 6</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>state 7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>state 8</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3: Transition Table with Equivalence Classes

<table>
<thead>
<tr>
<th>Equivalence Class</th>
<th>Other Letters</th>
<th>Non-Letters</th>
</tr>
</thead>
<tbody>
<tr>
<td>state 1</td>
<td>e</td>
<td>h</td>
</tr>
<tr>
<td>state 2</td>
<td>i</td>
<td>l</td>
</tr>
<tr>
<td>state 3</td>
<td>o</td>
<td>a</td>
</tr>
<tr>
<td>state 4</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>state 5</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>state 6</td>
<td>f</td>
<td>g</td>
</tr>
<tr>
<td>state 7</td>
<td>h</td>
<td>i</td>
</tr>
<tr>
<td>state 8</td>
<td>j</td>
<td>k</td>
</tr>
</tbody>
</table>

Table 4: Johnson's Representation

<table>
<thead>
<tr>
<th>Array Element</th>
<th>Base</th>
<th>Default</th>
<th>Next</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>-</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Figures 9 and 10 show the equivalent DFA. An example might clarify this somewhat elaborate diagram. If the characters “H”, “E”, and “L” have been read then the DFA will be in state #5. If it reads another “L”, it will enter state #6. If a letter other than “L” is read, the DFA will enter state #8 and will remain there as long as the input characters are letters. If a non-letter, such as a blank, is read while in state #5, the DFA will jam. Since state #5 is an accepting state for the catch-all ([A-Z]+) rule, the scanner will execute the action associated with that rule.

Table 2 clearly shows the redundancy of the DFA's transition table. The first step in compressing this table is to note that the 21 letters which are not specified in the patterns (that is, all except “E”, “H”, “I”, “L”, and “0”) are equivalent to one another (i.e., they have identical transitions) and can be grouped together. The 102 ASCII characters which are not letters can also be grouped together. We call these groups equivalence classes. In general, the members of an equivalence class all have identical entries in their column of the transition table. The rules in Figure 7 will result in seven equivalence classes. Equivalence class 1 will contain only the letter “E”, class 2 will contain only the letter “H”, and so forth. Equivalence class 6 will contain the remaining letters, and class 7 will contain all of the non-letters. The transition table can now be represented as shown in Table 3. A total of 968 elements (121 columns of 8 elements each) have been removed. With larger sets of rules the savings won't be as great, but there will typically be 40-60 equivalence classes, yielding a 50% to 70% reduction in size.

All of the “8”s left in Table 3 indicate that there is yet more redundancy. While the rows and columns are similar, few are identical, so we can't get a big savings by grouping more equivalent objects together. We can save more space by taking advantage of the similarities. S. C. Johnson of Bell Labs developed a way of representing the transition table which allows similar states to share data for common transitions. Figure 4 shows one way of representing the transition table shown in Table 3 using Johnson's scheme. The two pairs of arrays are used as follows. The Base/Default pair is indexed by the current state. The Base entry gives a base index. The equivalence

---

5 Our source for the representation was [Aho & Ullman], p. 116.
integer ecmap(MAXCHARS)

# Data to initialize ecmap, which maps characters
# to equivalence classes, go here.

ch = getc(ch)
ec = ecmap(ch)

while (Check(Base(state)+ec) != state)
    state = Default(state)

state = Next(Base(state)+ec)

For all of the values 1, 2, 3, and 5, Check(Base(3)+ec) = 3, so we then assign

    state ← Next(6+ec) = 8.

- if the character is not a letter then it is in equivalence class 7. Then

    Check(Base(state)+ec) = Check(1+7) = Check(8) = 3 ≠ 5.

so

    state ← Default(8) = 3.

Now

    Check(Base(3)+7) = Check(13) = 3

so we assign

    state ← Next(13) = JAN.

The transition table representation shown in Table 3 uses 56 array elements while the one in Figure 4 uses 42, not an impressive savings. The savings, however, become significant as the set of rules grows larger. There are many subtleties involved in getting a large amount of compression using Johnson’s representation which we hope to discuss in a future paper. Both UNIX Lex and Software Tools Lex use Johnson’s representation. The data tables they generate are summarized in Table 5 and the relative sizes shown in Figure 11. Occasionally, the second implementation of some piece of software improves on the original. We note in passing that the Software Tools Lex tables are smaller than the UNIX Lex tables in all cases and the reduction is more than 50% for large Lex inputs.
5. Conclusion

One problem with writing efficient programs is that it's hard to write efficient scanners. Software Tools Lex alleviates this problem by making scanner specification easy and by automatically writing scanners based on these specifications. Lex uses Finite Automaton techniques to generate fast scanners and Johnson's representation to make these scanners compact. The result is that Lex provides a powerful way of writing very efficient scanners.

6. Acknowledgements

The general design and early work on Lex was done by Jef Poskanzer. Van Jacobson contributed a great number of ideas and much feedback and inspiration throughout the entire development of Lex. Van Jacobson, Theresa Breckon, Marshall Spight, and John Lynch provided many helpful comments and suggestions concerning this paper.

7. References


---

**Table 5: Performance of Software Tools and UNIX Lex**

<table>
<thead>
<tr>
<th>Input</th>
<th># of ECs</th>
<th>Table Size</th>
<th>UNIX Lex</th>
<th>Tools Lex</th>
</tr>
</thead>
<tbody>
<tr>
<td>awk</td>
<td>69</td>
<td>14007</td>
<td>860</td>
<td>791</td>
</tr>
<tr>
<td>nwords</td>
<td>56</td>
<td>14224</td>
<td>3623</td>
<td>1420</td>
</tr>
<tr>
<td>ratfix</td>
<td>59</td>
<td>25370</td>
<td>3419</td>
<td>1648</td>
</tr>
<tr>
<td>sh</td>
<td>31</td>
<td>6665</td>
<td>2750</td>
<td>1097</td>
</tr>
<tr>
<td>typeof</td>
<td>31</td>
<td>7843</td>
<td>1145</td>
<td>822</td>
</tr>
</tbody>
</table>

"# of ECs" is the number of equivalence classes in the input. "Table Size" is equal to the number of DFA states times the number of equivalence classes. The columns for "UNIX Lex" and "Software Tools Lex" give the total number of array elements needed to represent the transition table using Johnson's representation.
This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.