The Language for Specifying Lexical Analyzer

We shall now study how to build a lexical analyzer from a specification of tokens in the form of a list of regular expressions. The discussion centers around the design of an existing tool called LEX, for automatically generating lexical analyzer program.

A LEX source program is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The action is a piece of code which is to be executed whenever a token specified by the corresponding regular expression is recognized. The output of LEX is a lexical analyzer program constructed from the LEX source specification.

Unlike most programming languages, a source program for LEX does not supply all the details of the intended computation. Rather, LEX itself supplies with its output a program that simulates a finite automaton. This program takes a transition table as data. The transition table is that portion of LEX's output that stems directly from LEX's input. The situation is shown in fig (23), where the lexical analyzer L is the transition table plus the program to simulate an arbitrary finite automaton expressed as a transition table. Only L is to be included in the compiler being built.

Fig (23) the Role of LEX
A LEX source program consists of two parts, a sequence of auxiliary definition followed by a sequence of translation rules.

**Auxiliary Definitions**

The auxiliary definitions are statements of the form:

\[
D_1 = R_1 \\
D_2 = R_2 \\
\vdots \\
D_n = R_n
\]

Where each \(D_i\) is a distinct name, and each \(R_i\) is a regular expression whose symbol are chosen from \(\Sigma \cup \{D_1, D_2, D_{i-1}\}\), i.e., characters or previously defined names. The \(D_i\)'s are shorthand names for regular expressions. \(\Sigma\) is our input symbol alphabet.

**Example:** We can define the class of identifiers for a typical programming language with the sequence of auxiliary definitions.

\[
\text{Letter} = A \mid B \mid \ldots \mid Z \\
\text{Digit} = 0 \mid 1 \mid \ldots \mid 9 \\
\text{Identifier} = \text{Letter} (\text{Letter} \mid \text{Digit})^*
\]
Translation Rules

The translation rules of a LEX program are statements of the form:-

\[
P_1 \{A_1\}
\]

\[
P_2 \{A_2\}
\]

\[
P_m \{A_m\}
\]

Where each \(P_i\) is a regular expression called a pattern, over the alphabet consisting of \(\Sigma\) and the auxiliary definition names. The patterns describe the form of the tokens. Each \(A_i\) is a program fragment describing what action the lexical analyzer should take when token \(P_i\) is found. The \(A_i\)'s are written in a conventional programming language, rather than any particular language, we use pseudo language. To create the lexical analyzer \(L\), each of the \(A_i\)'s must be compiled into machine code.

The lexical analyzer \(L\) created by LEX behaves in the following manner: \(L\) reads its input, one character at a time, until it has found the longest prefix of the input which matches one of the regular expressions, \(P_i\). Once \(L\) has found that prefix, \(L\) removes it from the input and places it in a buffer called \(TOKEN\). (Actually, \(TOKEN\) may be a pair of pointers to the beginning and end of the matched string in the input buffer itself.). \(L\) then executes the action \(A_i\).

It is possible, that none of the regular expressions denoting the tokens matches any prefix of the input. In that case, an error has occurred, and \(L\) transfers control to some error handling routine. It is also possible that two or more patterns match the same longest prefix of the remaining input. If that is
the case, L will break the tie in favor of that token which came first in the list of translation rules.

**Example:** Let us consider the collection of tokens defined in fig (5), LEX program is shown in fig (24)

**AUXILIARY DEFINITION**

Letter = A | B | … | Z

Digit = 0 | 1 | … | 9

**TRANSLATION RULES**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>{return 1}</td>
</tr>
<tr>
<td>END</td>
<td>{return 2}</td>
</tr>
<tr>
<td>IF</td>
<td>{return 3}</td>
</tr>
<tr>
<td>THEN</td>
<td>{return 4}</td>
</tr>
<tr>
<td>ELSE</td>
<td>{return 5}</td>
</tr>
<tr>
<td>letter(letter</td>
<td>digit)*</td>
</tr>
<tr>
<td>digit+</td>
<td>{LEX VAL:= INSTALL( ); return 7}</td>
</tr>
<tr>
<td>&lt;</td>
<td>{LEX VAL := 1; return 8}</td>
</tr>
<tr>
<td>&lt;=</td>
<td>{LEX VAL := 2; return 8}</td>
</tr>
<tr>
<td>=</td>
<td>{LEX VAL := 3; return 8}</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>{LEX VAL := 4; return 8}</td>
</tr>
<tr>
<td>&gt;</td>
<td>{LEX VAL := 5; return 8}</td>
</tr>
<tr>
<td>&gt;=</td>
<td>{LEX VAL := 6; return 8}</td>
</tr>
</tbody>
</table>

Fig (24) LEX Program.
Suppose the lexical analyzer resulting from the above rules is given input `BEGIN` followed by blank. Both the first and sixth pattern matches `BEGIN`, and no pattern matches a longer string. Since the pattern for keyword `BEGIN` precedes the pattern for identifier in the above list, the conflict is resolved in favor of the keyword.

For another example, suppose `<=` are the first two characters read. While pattern `<` matches the first character, it is not the longest pattern matching a prefix of the input. Thus LEX's strategy that the longest prefix matching a pattern is selected makes it easy to resolve the conflict between `<` and `<=`, by choosing `<=` as the next tokens.