Module 4- Lexical Analyzer Generator

This module discusses the core issues in designing a lexical analyzer generator from basis or using a tool. The basics of LEX tool are also discussed.

Need for a Tool

The lexical analysis phase of the compiler is machine independent. It comes under the analysis phase. The lexical analysis phase needs to tokenize the input string and hence it is source language dependent. The lexical analyzer needs to define patterns for all programming constructs of the input language. Hence, designing a lexical analyzer from the scratch is difficult. On the other hand, if there is a tool that can handle these variations in source language, then designing the lexical analysis phase would be easier.

Lexical Analyzer Generator Tool

Lexical Analyzer Generator is typically implemented using a tool. There are some standard tools available in the UNIX environment. Some of the standard tools are

- LEX – it helps in writing programs whose flow of control is regulated by the various definitions of regular expressions in the input stream. The wrapper programming language is C.
- FLEX – It is a faster lexical analyzer tool. This is also a C language version of the LEX tool.
- JLEX – This is a Java version of LEX tool.

The control flow of a lex program is given in Figure 8.1.

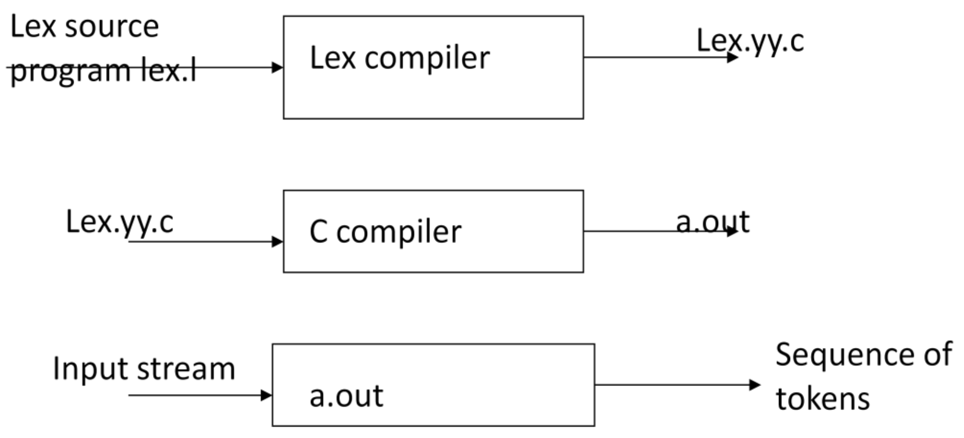

Figure 8.1 Control flow of a lex program

The input will be a source file with a ―.l‖ extension. This will be compiled by a Lex compiler and the output of this compiler will be a source file in C named as ―lex.yy.c‖. This can be compiled using a C compiler to get the desired output.

Components of a LEX program

A LEX program typically consists of three parts: An initial declaration section, a middle set of translation rules and the last section that consists of other auxiliary procedures. The ―%%‖ acts as a delimiter which separates the declaration section from the translation rules section and the translation rules section from the auxiliary procedures section. A program may miss the declaration section but the delimiter is mandatory.

```
declaration

%%

translation rules

%%

auxiliary procedures
```

In the declaration section, declarations and initialization of variables take place. In the translation rules section, regular expressions to match tokens along with the necessary actions are defined. The auxiliary procedures section consists of a main function corresponding to a C program and any other functions that are required in the auxiliary procedures section.

Declaration

In the declaration section a regular expression can be defined. Following is an example of declaration section. Each statement has two components: a name and a regular expression that is used to denote the name.

```
1. delim [\t\n]
2. ws     {delim}+
3. letter [A-Za-z]
4. digit   [0-9]
5. id       {letter}({letter}|{digit})*
```

Table 8.1 summarizes the operators and special characters used in the regular expressions which are part of the declaration and translation rules section.

Table 8.1 Meta Characters

Meta Character	**Match**
.	Any character except new line
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[ ]	character class

In addition, the declaration section may also contain some local variable declarations and definitions which can be modified in the subsequent sections.

Translation Rules

This is the second section of the LEX program after the declarations. The declarations section is separated from the Translation Rules section by means of the ―%%‖ delimiter. Here, each statement consists of two components: a pattern and an action. The pattern is matched with the input. If there is a match of pattern, the action listed against the pattern is carried out. Thus the LEX tool can be looked upon as a rule based programming language. The following is an example of patterns p_1, $p_2 \ldots p_n$ and their corresponding actions 1 to n.

p_1 {$action_1$} /*p—pattern (Regular exp) */

…

p_n {$action_n$}

For example, if the keyword IF is to be returned as a token for a match with the input string ―if‖ then the translation rule is defined as

{if} {return(IF);}

The ―;‖ at the end of the (IF) indicates end of the first statement of an action and the entire sequence of actions is available between a pair of parenthesis. If the action has been written in multiple lines then the continuation character needs to be used. Similarly the following is an example for an identifier ―id‖, where the usage of ―id‖ is already stated in the first ―declaration‖ section.

```
{id}  {yylval=install_id();return(ID);}
```

In the above statement, when encountering an identifier, two actions need to be taken. The first one is call install_id() function and assign it to yylval and the second one is a return statement.

Auxiliary procedures

This section is separated from the translation rules section using the delimiter –%%‖. In this section, the C program's main function is declared and the other necessary functions also defined. In the example defined in translation rules section, the function install_id() is a procedure used to install the lexeme, whose first character is pointed by yytext and length is provided by yyleng which are inserted into the symbol table and return a pointer pointing to the beginning of the lexeme.

```
install_id() {

            }
```

The functionality of install_id can be written separately or combined with the main function. The functions yytext ( ) and yyleng( ) are lex commands to indicate the text of the input and the length of the string.

Example LEX program

The following LEX program is used to count the number of lines in the input data stream.

```
1. int num_lines = 0;
2. %%
3.   \n      ++num_lines;
4.   .       ;
5.  %%
6.   main()
7.   {  yylex();
8.   printf( "# of lines = %d\n", num_lines);  }
```

Example 8.1 LEX program to count the number of lines

Line 1 of this program belongs to the declaration section and declares an integer variable num_lines and initializes it to 0 and this concludes the first section. In the translation rules section the pattern in –\n‖ which is defined in line 3 needs to be matched with the action, incrementing the variable num_lines. This statement indicates whenever a new line is encountered which is defined by \n, the line number is incremented. The third section is from line numbers 6 to 8. yylex( ) points to the text defined in the input stream and as long as data exists in the input, the line numbers are counted and printed in Line 8.

Example 8.2 is an extension of example 8.1 where number of characters, words and lines are counted.

```
1. %{ int nchar, nword, nline; %}
2. %%
3.   \n { nline++; nchar++; }
4.   [^ \t\n]+ { nword++, nchar += yyleng; } . { nchar++; }
5. %%
6.   int main(void)
7.   { yylex();
8.   printf("%d\t%d\t%d\n", nchar, nword, nline);
9.   return 0; }
```

Example 8.2 LEX program to count number of lines, characters and words

In this example –\n‖ is mapped to increment the number of characters and number of lines, while \t, a tab space is used to increment the number of words and number of characters which is defined in line numbers 3 and 4. Lines 6 to 9 represent the main function, which uses yylex to point to the text and processes it to count the number of lines, characters and words.

Table 8.2 is a summary of some of the yy() commands that can be used in LEX program.

Table 8.2

Name	**Function**
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string

The syntax and compilation procedure of FLEX program is the same as that of the LEX program.

JLEX program

A typical JLEX program also consists of three parts. In the case of JLEX program, the organization is slightly different. The first section represents the user code which is later copied directly as a Java file. The second section consists of JLEX directives, where macros and other state names are typically defined. Here again, –%%‖ is used as a delimiter to separate one section from the other. The third section consists of the translation rules that define regular expressions along with the necessary actions. The user code is copied to a java file and a JAVA compiler is used instead of a C compiler of the LEX program to compile and execute the JLEX program. The following is a simple layout of a JLEX file.

User code

%%

JLex directives

%%

Lexical analysis rules

Example 8.3 is a JLEX program to count the number of lines in an input document.

```
1.  import java_cup.runtime.*;
2. %%
3.  %cup
4.  %{
5. private int lineCounter = 0;
6. %}
7.  %eofval{
8.  System.out.println("line number=" + lineCounter);
9.  return new Symbol(sym.EOF);
10. %eofval}
11. NEWLINE=\n
12. %%
13. {NEWLINE} {
14. lineCounter++;
15. }
16. [^{NEWLINE}] { }
```

Example8.3 Sample JLEX program

In example 8.3, line number 1 indicates the user code which gets copied to the java file. %cup is used to activate the Java CUP compatibility that helps conform to the java_cup.runtime and initiates the scanner. The Java CUP compatibility is turned off by default and hence every JLEX program needs to get activate as the first step. %eofval is used to indicate –end of value‖ of the input. The variable line number is incremented for every NEWLINE encountered in line number 14. NEWLINE is a name given to the –\n‖ character which indicates the encounter of a new line. This is again a rule based programming language and hence does not obey the flow of a structured programming language.

Summary

This module discussed the use of tools LEX and JLEX for tokenizing the input. Designing and implementing the lexical phase of the compiler is difficult and hence tools can be used to do the job of tokenizing using a rule based programming language. The tools use regular expressions to define the pattern and a corresponding action. When the regular expression matches a pattern, the defined action is used to perform a task.