4

Theory of FLEX

This chapter discusses the theory and format of the Lex specifications and describes the features and options available. It also explains the theoretical aspects of Flex syntax and semantics that summarizes the capabilities of the Flex specifications demonstrated in the previous chapters.

4.1 STRUCTURE OF A FLEX PROGRAM

As we had discussed earlier, a Lex program consists of three sections—the definition section, rules section, and auxiliary section (or user defined section or user subroutines). They are separated by a line consisting of two percent signs, %%.

```
Definition section
%%
Rules section
%%
Auxiliary section
```

The first two parts are necessary, even though they are empty. The third part and the preceding %% line may be omitted.

4.1.1 Definition Section

The definition section of a Lex program can be empty. This may contain the literal block, definitions, internal table declaration, variable name definitions, start conditions and translations for both C and Lex statements and functions. They also create an atmosphere in which the Lex program can work more efficiently. The declaration and definitions attached to the C statements are separated by "%{" and "}%", and they are copied verbatim to the C file, which is a lexical analyzer; *lex.yy.c* (by default).

4.1.2 Rules Section

The rule section contains patterns and actions C code. That is, a rule contains a pattern (which is a regular expression), followed by a white space and C code, which will be executed when the input stream matches the pattern. If the C code is more than one statement or spaces or multiple lines, it must be placed within braces "{" and "}" to show that it is a single block of statements. In the absence of braces, some versions of Lex take the entire rule of the statement line; others just take up to the first semicolon (*i.e. first statement only*).

When we execute the lexical analyzer, it matches the input stream against the patterns in the rule sections, and if it is so, it executes the corresponding C code associated with that pattern. If the matched pattern is followed by a single vertical bar, instead of action C code, the matched pattern uses the same action code as that of the next pattern. When one input character does not match with any patterns, it is simply copied into the output (i.e. standard output or stdout).

4.1.3 Auxiliary Section

The auxiliary section contains user defined C function (subroutines, including the main() function, from where the execution begins). They are copied as it is to the lexical analyzer C file by Flex. In this section you can redefine your own versions of *input()*, *unput()*, *output*, *yymore*, etc.

4.2 PATTERN SUBSTITUTION

The pattern definition allows you to give a name to all or part of a regular expression, to refer and substitute by the given name in the rules section. This can be useful to break up complex expressions and to document them. For example, in Program 4.1 DIGIT is declared as [0–9] in the definition section, which will be replaced in the rules section wherever DIGIT is found in the pattern.

PROGRAM 4.1

```
------------------------------------------------------------
DIGIT [0-9]
Alphabet [A-Z a-z]
OPERATOR [+ | \- | / | *]
--------------------------
--------------------------
%%
{DIGIT}+          { Integer_Action();     }
{Alphabet}+    { Character_Action();   }
{OPERATOR}+    { Operator_Action(); }
------------------------------------------------------------
```

4.3 BEGIN

The BEGIN macro switches among start states. It has the structure *BEGIN<start symbol>*. We can invoke any start state patterns just by calling the same using BEGIN.

When the scanner starts scanning, the lexer is in a state 0 (zero), also known as INITIAL state. At any point of execution, we can go back to the initial state by just invoking BEGIN INITIAL.

All the other start symbols (other than INITIAL) must be declared in the line along with %s or %x in the definition section of the Flex program.

Note: Even though BEGIN is a macro, and does not take any argument itself, the start symbol name need not be enclosed in parentheses. But it is always a good practice to do so.

4.4 ECHO

The macro ECHO, in the action C code of any pattern, writes the recently matched token *to the current output file yyout.* Program 4.2 explains the same.

PROGRAM 4.2

```
---------------------------------------------------------------------
%%
[a-zA-Z]+ { printf("\nThe token is (using ECHO) = "); ECHO;
            fprintf(yyout, "The token is (using fprintf) = %s", yytext);
  }
%%
main()
{
  yylex();
}
---------------------------------- output ----------------------------------
Flex

The token is (using ECHO) = Flex
The token is (using fprintf) = Flex
---------------------------------------------------------------------
```

As is explained in the above program, ECHO is equivalent to

```
fprintf(yyout, "%s", yytext);
```

In some versions of Lex, we can redefine ECHO, to do something else with recently matched token characters. Along with redefining ECHO, we should also have to redefine the function *output()*, which normally sends as single character to *yyout*.

4.5 REJECT

The directive REJECT directs the scanner to proceed to the second best rule to match the prefix of the input. That is, when an action REJECT executes REJECT, Lex conceptually puts back the text matched by the pattern and finds the next best match for it. Program 3.8 explains the REJECT concept.

Lexical scanners that use REJECT may be much larger and slower than those that do not, since they need considerable extra information to allow backtracking and re-lexing.

4.6 START STATE

We can declare start states, also called *start conditions* and *start rules,* in the definition section of the Flex program. They are used to limit the scope or life-time of certain rules. This will change the way the lexer tracks some parts of the file. That is, we can use a start state to apply a set of rules only at certain time. Flex provides two types of start conditions, which are discussed in Section 3.7. Flex has exclusive start states declared using unintended line, beginning with %x and inclusive start states with %s.

Note that these rules that do not have start states will be active throughout the execution of the lexer, and it can be applied from any state.

4.7 LEFT AND RIGHT CONTEXT

There are several ways to make a pattern sensitive to the left and right context. Lex provides several methods to give higher or lower precedence to the patterns that precede or follow the token.

Left Context

There are three ways to handle the left context, which will give different precedence to different rules.

The first method that we use to make a pattern left context is to have some special character at the beginning of the pattern. That is, the character "^" at the beginning of a pattern tells Flex to match the pattern only at the beginning of the line, which does not match any character that is just followed in the context. For example, the pattern [^a-z] will match any character other than the lower case letter 'a to z'.

The second way to make a pattern precede another is to use start states. We can activate or disable a pattern by using start state as we need one token to precede another. This is shown in Program 4.3.

PROGRAM 4.3

```
------------------------------------------------------------------

%s START1 START2

#    { BEGIN START1; }
## { BEGIN START2; }
-------------------------
-------------------------
<START1> [a-z]+  {  statement 1
                      statement 2
                      ------------------
                      ------------------
                      statement n
                      BEGIN 0;
                    }

<START2> [0-9]+ {   statement 1
                      statement 2
                      ------------------
                      ------------------
                      statement n
                      BEGIN INITIAL;
                    }
------------------------------------------------------------------
```

Refer to Section 3.7; for more explanation on Program 4.3.

In some case you can have explicit code (same as we write C code) for specifying the left context sensitivity by setting different values to different variables to pass from one pattern to another. Program 4.4 shows the same.

PROGRAM 4.4

```
------------------------------------------------------------------

%{
    int flag = -1
%}

%%

[a-z]+ { flag = 0; }
[A-Z]+ { flag =1; }
[0-9]+ { flag = 2; }
ACTION { switch(flag)
```

```
        {
            case 0: lower_case_token();
                    break;
            case 1: upper_case_token();
                    break;
            case 2: digit_token();
                    break;
            default:
                    break;
        }
        flag = -1
    }
```

Right Context

Lex provides three ways to make the pattern sensitive to the right context, that is, to the text to the right of the token.

The first method that we use to make a pattern right context is to have special character at the end of a pattern. The '$' character at the end of a pattern makes the token match only at the end of a line (i.e. immediately before a \n character). Like '^' character, '$' does not match any character, it just specifies the context.

The second method is slash operator. The '/' character in a pattern allows you to include the explicit trailing context. For example, the pattern *abc/xyz* matches the token *abc*, only if it is immediately followed by *xyz*. The trailing context characters (i.e. *xyz*) do not appear in *yytext*, nor are counted in *yyleng*.

The *yyless()* function is another method to make the pattern right context sensitive. It pushes back part of the token to the input stream that was just read. For example,

```
abcdef { yyless(3); }
```

It returns all characters, except first three characters (i.e. *abc*) of the current token (i.e. *abcdef*) back to the input stream, where they will be re-scanned when the scanner looks for the next match.

4.8 FLEX INPUT()

The lexical analyzer reads from the input file (by default it is *yyin*), which is a standard input (i.e. *stdin*) file. The input() function is used to read from the source file to get characters and strings as it is in the memory.

The input() function provides characters to the lexer when the scanner matches characters. It conceptually calls input() to fetch each character from the input file. Program 3.14 explains how the input() processes the characters until either end-of-file or the characters "*/" occur to handle C comments.

Although Flex provides an input() function, it gets characters using optimized-in-line code. We can redefine YY-INPUT, a macro used to read blocks of data, to read from any input file. It is of the form

```
YY_INPUT(buffer, result, max-size)
```

Where *buffer* is a character buffer, *result* is a variable to store the number of characters read, and Max_Size is the maximum size of the buffer where string characters are read into. We can redefine YY_INPUT for reading from a string buffer as follows. Program 4.5 shows how a YYINPUT can be used to read a block of input datas.

PROGRAM 4.5

```
---------------------------------------------------------------
%{
     #undef YY_INPUT
     #define YY_INPUT(bu,re,ms)
     {
        re = my_yyinput(bu,ms)
     }
%}

---------------------------------
---------------------------------

extern char muinput[];
/* myInputPtr is a pointer variable that points to current position of the given
input file */
extern char *myInputPtr;
extern int *myInputLim;

-----------------------------
-----------------------------

int mu_yyinput (char*buffer, int max_size)
{
  int n = min(max_size, myInputLim, myInputPtr);
  if(n>0)
  {
    memoryCopy(buffer, myInputPtr,n);
    myInputPtr +=n;
  }
  return n;
}
---------------------------------------------------------------
```

4.9 FLEX OUTPUT()

Flex defines a function *output(c)*, in flex library, which writes its argument to the output file *yyout*. This is equivalent to

```
putc(c,yyout);
```

The *output()* function can be used along with the action C code. It can be redefined to match all possible inputs. If we redefine *output()*, then we must also redefine the macro ECHO which copies the current contents of *yytext* to the output.

4.10 REGULAR EXPRESSION

The rule section of a Flex program contains the number of rules, which match against the given input stream. Each rule contains the pattern and the corresponding action statements (i.e. C code). Lex patterns are an extended version of the regular expressions used by the editors and utilities such as 'grep' in Linux. Regular expressions are composed of normal characters, which represent themselves and meta-character which have a special meaning in a pattern. All characters other than those listed in Section 2.7 are regular characters.

4.11 FLEX LIBRARY FUNCTIONS

Many programming languages have *include* (i.e. #include) statements that logically insert another file in place of the include statement. But unfortunately, there is no way in Flex to handle the inclusion of another input files, except assigning the input file to *yyin* to have scanner to read from that file.

When a lexer reaches the end of the input file, it calls *yywrap()*. If *yywrap()* returns false (i.e. zero), then it is assumed that the function has gone ahead and set up *yyin* to point to another input file, and scanning continues and if it returns true (i.e. non-zero) then the scanner terminates returning 0 to its caller.

But we can write our own *yywrap()* that switches to a new input file by changing or re-opening *yyin*, and continues scanning. See Program 3.18 for more details on *yywrap()*.

Like *yywrap()*, Flex comes with a number of functions and macros; they come along with Flex libraries. We can link these libraries by giving the—*lfl* flag at the end of *cc* command line.

All the Flex programs should have (minimum) *main*() functions, from where the execution begins. Main() function is defined in the auxiliary section of the Lex program, by the Flex tool if the user is not providing one. The user can redefine the main() function at the auxiliary section.

4.11.1 unput()

The function *unput(a)* puts or returns the character *a* into the input stream and it will be the next character to be scanned. Even though we can call *unput()* function several times in a row to push several characters back to the input, the limit of data pushed back by *unput()* varies. It depends on how it is defined in library; but it is always at least as great as the largest token the lexer recognizes.

Some versions of Lex allow us to redefine *unput()* to change the scanners input to handle multiple push back characters. The unput function is further explained in Flex Program 3.13.

4.11.2 yyinput(), yyoutput(), yyunput()

Flex provides the function *yyinput(), yyoutput()* and *yyunput()* as wrappers for the macro input(), output() and unput(), respectively. They can be redefined by the user in the auxiliary section and can be called from other modules such as the lex library.

4.11.3 yytext and yyleng

The lexical analyzer matches the input stream with rules to identify tokens. Whenever the scanner matches a token, the text of the token is stored in the null terminated string *yytext* and its length in *yyleng*. The length in *yyleng* is the same as the value returned by *strlen(yytext)*.

The *yytext* is a character variable that can be declared as an array or pointer variable

```
  exterm char yytext[];
  exterm char *yytext;
```

The contents of the *yytext* are replaced as and when the new token is found and matched by a rule. If the size of the token is larger than the size of the *yytext* array, then an overflow occurs. In Flex the standard size for *yytext[]* is 200 characters. Even if *yytext* is a pointer, the pointer points into an I/O buffer, which is also of limited size, and similar problems can arise from very large tokens. In Flex the default I/O buffer is 10K, and it can handle tokens up to 8K, which is certainly large enough.

Imagine a situation, where we need to handle bigger tokens of size greater than 8K bytes, and then we have the option to redefine buffer size if the memory space is available. Flex buffers are created by a function called *yy_create_buffer()*, and *yy_current_buffer* points to the current buffer (which is active). We can create a buffer of any size as follows.

```
%%
setupbuf(int size)
{
  yy_current_buffer= yy_create_buffer(yyin,size)
}
```

The technique for increasing the buffer size differs as the lex version differs.

4.11.4 yyless()

We can call *yyless(n)* from the C code associated with a rule to return or put all characters except first *n* characters to the input stream, where they will be re-scanned when the scanner looks for the next match. The *yytext* and *yyleng* is adjusted approximately. Section 3.8.2 explains the use of *yyless()* in detail.

Note that a call to *yyless()* has the same effect as calling *unput()* with the characters to be pushed back, but *yyless()* is often faster because it can take advantage of the fact that the characters pushed back are the same ones just fetched from the input.

4.11.5 yylex()

yylex(), an entry point to the scanner, is a function created by the Flex from the rules section. All C codes in the rules section are copied into *yylex*. We can call *yylex()* to start or resume scanning. If any Lex action executes return statement to pass a value to the calling function, the next call to *yylex()* will continue or restart from the point where it left off.

Returning values from *yylex()* have got wide use in parser, for example, identifying and returning valid tokens such as keyword, variable name or operator of the parser's interest from the lexer. And when it matches a token not of interest (i.e. white space or a comment) to the parser, it does not return and the scanner immediately proceeds to match other tokens by restarting from the point where it left off. We cannot restart a lexer just by calling *yylex()* again. Instead, we have to reset it into the default state using BEGIN INITIAL and discard all the input buffered by *unput()*, and so on. But Flex makes restarting a lexer considerably easier by calling *yyrestart*(file), and when the file is a standard I/O file pointer, arranges to start reading from that file.

4.12 MULTIPLE LEXERS IN ONE PROGRAM

We can have two different lexers, meant to identify different types of token, with an entirely different syntax in one file or in different files. This concept is mainly used in interactive debugging interpreter, where we need one lexer for the programming language and the other for debugging statements.

There are two basic approaches for handling two lexers in one program:

- Combine them into a single lexer
- Put them (two complete lexers) in a separate file

4.12.1 Combined Lexers

We can combine two lexers to one by using start states. All the patterns in a rule sections are prefixed by a unique set of start states, which help us to enable or disable any rule conditionally. That is, a few rules are enabled when a start state calls with special directive BEGIN, which constitutes a lexer. When another condition satisfies, or else when we want to switch on to another lexer, BEGIN calls start symbol, which will enable another set of rules (i.e. another lexer) by disabling the earlier rules, and so on.

The main advantage of this approach is that, different lexers can share rules and other object codes. That is, a rule will remain to stay active in some or all lexers, when they are defined with start symbols. So the object code is somewhat smaller since there is only one copy of the lexer code and the different rule sets can share the same rules. The disadvantage is that, we have to be very careful to use the correct start status everywhere, because we cannot have both the lexers active simultaneously.

4.12.2 Multiple Lexers

In multiple lexers, we have different lexers in different files with different entry points, and later club them all into one file to call it.

Multiple lexer is an approach to include two complete different lexers in one program. This is not easy to implement, because every lexer generated by the Flex has the same entry point: *yylex()*. Moreover, the scanning table, scanner buffers and other functions are like global variables and their scope is throughout the program execution. To avoid duplicate variable and function calling and others, it is better that we change the names that Flex uses for its functions and variables.

Flex provides a command-line switch '-P' to change the prefix used on the names in the scanner generated by Lex (see the option -P in Section 4.13); that is, using the "P" flag we can change the names that Lex uses in two scanners, which is actually there in one program file. For example,

```
flex-Pmul multiply.l
```

produces a scanner with the entry point *mullex()*, which reads from file *mulin* and so forth. The names affected are *yylex()*, *yyin*, *yyout*, *yytext*, *yylineno*, *yyleng*, *yymore()*, *yyless()*, *yywrap()*, as well as all of the implementation specific variables.

There is no other method, but to fake the name of the generated C functions. The most easier way to fake it, create the file *yy_sed* containing following *sed* commands, and use the stream editor command *sed* (here we use a prefix *mul*).

```
s/yyback/mulback/g
s/yybegin/mulbegin/g
s/yycrack/
s/yyerror/
s/yyestate/
s/yyextra/
s/yyfind/
s/yyin/
s/yyinput/mulinput/g
s/yyleng
s/yylex
s/yylineno
s/yy/sp
s/yylook
```

```
s/yy/val
s/yylstate/mul/state/g
s/ymatch/
s/yy/morefg/
s/yyout/
s/yyolsp/mulolsp/g
s/yyoutput/
s/yypreviour/
s/yystart/
s/yysptr/
s/yysvec/
s/yytchar/
s/yytext/
s/yytop/
s/yyunput/
s/yyvatop/
s/yywrap/mulwrap/g
```

After we have created scanner (i.e. *lex.yy.c*), the following command is executed.

```
sed -f yy_sed lex.yy.c > lex.mul.c
```

Another approach that will help us to avoid using *sed* is to use C preprocessor *#define* at the beginning of the grammar to rename the variables:

```
%{
        #define        yyback        mulback
        #define        yybgin        mulbgin
        #define        yycrack       mulcrack
        #define        yyerror       mulerror
        #define        yyestate      mulestate
        #define        yyextra          "
        #define        yyfind           "
        #define        yyin
        #define        yyinput
        #define        yyleng           "
        #define        yylex
        #define        yylineno
        #define        yylook
        #define        yylsp
        #define        yylstate         "
        #define        yylval
%}
```

For Flex lexers, the variables that need to be renamed are

```
yy_create_buffer
yy_delete_buffer
yy_init_buffer
yy_load_buffer_state
yy_switch_to _buffer
yyin
yyleng
yylex
yyout
yyrestart
yytext
```

You can use any one of above techniques to rename them.

As we have discussed earlier, one Flex program will be written and the lexer will be generated normally with an entry point *yylex()*. The second lexer is another and generated with an entry point *mullex()* (or *<prefix name>lex()*).

Generally in Flex, we define or include the token specification of the parser as the normal C preprocessor statement *#include<y.tab.h>,* which will be discussed in Chapter 6. But as far as the prefixed lexer is concerned, we include the lexer as *#include<mul.tab.h>*. Thus, we can include or combine multiple lexers (which are in different files) into one file to access from that program.

4.13 OPTIONS AVAILABLE IN FLEX

Flex has the following options to make a lexical generation more efficient.

-b This option generates backing up information to *lex.backup*. This is a list of scanner states which require backup. They are called backing up states. If we eliminate all backing up states, then the generated scanner will run faster.

-d This option makes the generated scanner run in the debug mode.

-f This option specifies fast scanner. No table compression is done and *stdio* is bypassed.

-h It generates the summary of Flex's options (i.e. help) to *stdout* file before it exits.

-i This option instructs Flex to generate a case insensitive scanner. The case of letters given in the Flex input patterns will be ignored and tokens in the input will be matched regardless of the case. The matched token will be made available to *yytext* and the case sensitive will be preserved.

-p This generates a performance report of the Lex program to *stderr*. The report contains comments regarding features of the Flex input file, which will cause a serious loss of performance in the resulting scanner.

-t It instructs the Flex to write the scanner that generates to standard output instead of *lex.yy.c.*

-w This option suppresses the warning messages.

-s It will suppress the default rule. Generally, unmatched inputs are verbatim copied to the output (e.g. echoed to stdout).

-J It instructs Flex to generate an interactive scanner. An interactive scanner is one that only looks ahead to decide what token has been matched.

-B This option instructs Flex to generate a batch scanner, the opposite of interactive scanners generated by "-I".

-V It prints the version number to *stdout* and exits.

-7 This option tells Flex to generate a 7-bit scanner. That is, one which can only recognize 7-bit characters in its input. The advantage of using '-7' is that scanners table can be up to half the size those generated using the '-8' options (see below). The disadvantage is that such scanners hang or crash if their input contains an 8-bit character.

-8 It instructs Flex to generate an 8-bit scanner, i.e. one which can recognize 8-bit characters.

-o<output file> It directs Flex to write the scanner to the given output file instead *of lex.yy.c.*

-P<prefix name> This option changes default *yy* prefix used by Flex for all globally visible variable and function names to the given new prefix name. For example, -Pmul changes the name of *yytext* to *multext*. It also changes the name of the default output file from *lex.yy.c* to *lex.mul.c*. Note that this option will help us to link multiple Flex programs together into the same executable. Since we are using a renamed *yymap()*, we must either provide our own version of redefined subroutine or use *%option noyywrap*. Because as we link with *-lfl*, Flex no longer provides any library functions to execute.

-s<skeleton-file> This option overrides the default skeleton file from which Flex constructs its scanners. We should never use this option unless we are doing Flex maintenance or development. All these Flex options can be used along with lexical analyzer generation command-line as follows.

flex <options> <flex program name> Flex also provides a mechanism for controlling options within the scanner specification itself, rather than from the Flex command line. This is done by including *%option* directive in the first section of the scanner specifications. We can specify multiple options with a single %option directive or multiple directives.

4.14 GENERATING C++ SCANNERS

Flex provides two different ways to generate C++ scanners. The first way is to simply compile a scanner generated by Flex using a C++ compiler instead of a C compiler. We have to use C++ code in the action part of the rules section instead of C code, to avoid any compilation errors. Remember that the default input source for the scanner remains *yyin*, and the default standard output is echoed to *yyout*. These variables remain "file *" file pointer variable, not the C++ streams.

The second method that Flex provides to generate C++ scanner is to use the option '-+' (equivalent to use %option C++), which is automatically specified if the name of the Flex execution ends in a '+', such as flex++. When we use this option, the generated scanner will write to the file *lex.yy.cc* instead of *lex.yy.c*. The generated scanner includes the header file *FlexLexer.h*, which defines the interface to two C++ classes.

The first class, FlexLexer, provides an abstract base class defining the general scanner class interface. The second class *yyFlexLexer*, which is defined from *FlexLexer*, defines additional member functions and protected virtual functions that can be redefined by the user. This book restricts itself to C++ scanners generation.