3

PROGRAMMING WITH FLEX

In this chapter, we discuss how to program in Flex. Flex is a software tool for building lexical analyzers or lexers. A lexical analyzer takes input streams and tokenizes it, i.e. divides up into lexical tokens. This division into units (which are usually called *tokens*) is known as *lexical analysis* or *lexing* in short. Flex takes a set of rules for valid tokens and produces a C program which we call a lexical analyzer or a lexer or a scanner in short, that can identify these tokens. The set of rules or descriptions you give to Flex is called a *Flex specification*.

The token descriptions or rules that Flex uses are known as *regular expressions*. Flex turns these regular expressions into the lexer that scans the input text and identifies the tokens. The lexer generated by Flex is almost always faster than a lexer that you might write in C by hand. This chapter deals with the syntax and semantics of Flex tool to generate the lexical analyzer, and the terms Lex and Flex are interchangeably used to represent the same.

3.1 FLEX

Flex is a fast lexical analyzer generator—a tool for programming that recognizes lexical patterns in the input with the help of Flex specifications. Flex specification contains two parts: (i) patterns and (ii) corresponding action. When you write a Flex specification, you create a set of patterns which the lexer matches against the input. Each time one of the patterns matches, the corresponding action part is invoked (which is a C code). In this way, a Lex program divides the input into tokens.

Flex itself does not produce an executable program; instead, it translates the Lex specifications into a file containing a C subroutine called *yylex()*. More precisely, all the rules in the rules section will automatically be converted into the C statements by the Flex tool and will be put under the function name of *yylex()*. That is, whenever we call the function *yylex*, C statements corresponding to the rules will be executed. That is, we call *yylex()* to run the lexer. The generated lexical analyzer, by default it is *lex.yy.c*, can be compiled using regular C compiler along with any other files and Flex libraries you want.

The program that you write in a Lex program contains the Lex specification and other C statements and subroutines. This file is named with an extension *<filename>.l* (for example, *file.l*). When this Lex program is passed to the Flex, it translates the *<filename>.l* into a file named *lex.yy.c*, which is a C program. Figure 3.1 shows the phases of a Lexical analyzer.

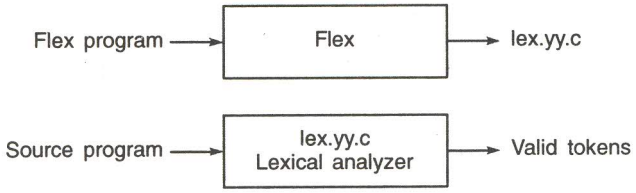

FIGURE 3.1 Phases of a Lexical Analyzer.

The file *lex.yy.c* is also called the *lexer* or *lexical analyzer*. This C program is compiled with a normal C compiler and it produces an executable program. This can be executed like a normal C executable file.

3.2 STRUCTURE OF FLEX PROGRAM

Any Flex program consists of three sections separated by a line with just %% in it.

```
Definition Section
%%
Rules Section
%%
User code (Auxiliary) Section
```

Let us now look at the basic structure of a Lex program. It consists of three sections.

1. A definition section
2. A rules section
3. A user defined section

The first section, i.e. the definition section contains different user defined Lex options used by the lexer. It also creates an environment for the execution of the Lex program.

The definition section helps to create an atmosphere in two areas. First, it creates an environment for the lexer, which is a C code. This area of the Lex specification is separated by "%{" and "%}", and it contains C statements, such as global declarations, commands, including library files and other declarations, which will be copied to the lexical analyzer (i.e. *lex.yy.c*) when it is passed through the Flex tool. In other words, Flex copies all the statements in this C declaration section bracketed by "%{" and "%}" to the lexical analyzer file, which is *lex.yy.c*.

Secondly, the definition section provides an environment for the Flex tool to convert the Lex specifications correctly and efficiently to a lexical analyzer. This section mainly contains declarations of simple name definitions to simplify the scanner specifications and declarations of start condition. The statements in this section will help the Lex rules to run efficiently.

The second section of any Lex program is the rules section that contains the patterns and actions that specify the Lex specifications. A pattern is in the form of a regular expression to match the largest possible string. Once the pattern is matched, the corresponding action part is invoked. The action part contains normal C language statements(s). They are enclosed within braces (i.e. "{" and "}"), if there is more than one statement, to make these component statements into a single block of statement.

Most versions of the Lex tools take everything after the pattern to be the action, while others only read the first statement on the line and silently ignore other statements. Always use braces to make the code clear, if the action has more than one statement or more than one line large.

The lexer always tries to match the largest possible string, but when there are two possible rules that match the same length, the lexer uses the first rule in the Lex specification to invoke its corresponding action.

The third and the final section of the Lex program is the user defined or user subroutine section. It is also known as *auxiliary section*. This section contains any valid C code. Flex copies the contents of this section into the generated lexical analyzer as it is. The simplest Flex program is

```
%%
```

which generates a scanner that simply copies its input (one character at a time) to its output. The following Flex programs will explain the syntax and semantics of the Flex specifications.

PROGRAM 3.1

Before we discuss any specific features of Lex, let us look at some simple Lex programs and analyze and understand how they work. Consider a simple Lex program given below, Program 3.1, which prints *Hi Good Morning* as we press the 'Enter' key.

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

//*************************************************************************
//FLEX program that Program to show the message
//when an ENTER key is pressed
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
```

```
%{
/*Program to show the message when an
        ENTER key is pressed */
%}

%%

[\n] { /* Display the following message */
          printf ("\n\nHi...Good Morning...\n");
       }

%%


main()
{
yylex();
}
```

Definition

Rules

Auxiliary procedure

In the first section, definition description of the program is shown. The whole part of this section will be copied to the lexical analyzer as it is.

The second section, i.e. rules section, contains the pattern and corresponding action. The pattern part of a Lex specification contains the regular expression of the lexical analyzer and the second part contains the action part, which is a C code, and will be carried out when a pattern matches with the input strings. In the above program, the pattern is "[\n]", which is an Enter key or a New line. Whenever the input string matches with the above said pattern string, statements in the action block, which are in the open curly braces ("{") and closing curly braces ("}"), will be executed. Here only one statement (*printf("\n\n Hi....Good Morning.....\n");* will be executed. That is, whenever an Enter key is pressed or a new line is found, it will display or print *Hi.....Good Morning....*

Note that when this Lex program is passed through the Flex tool to generate the lexical analyzer, it will convert the rules section into a C function, named *yylex()* automatically, which enables us to call from the main function. That is why we called the function *yylex()* in the main function, even though we have not defined it anywhere in the program. (Normally in C language we have to define a function before it is called.)

3.3 TRANSLATING, COMPILING AND EXECUTING A FLEX PROGRAM

Let Program 3.1 be in a file called *program.l*. To create or generate a lexical analyzer we must enter the following commands.

```
[root@Zion testFlex]#flex program.l  (i.e. flex <filename.l> )
```

When the above command is executed, Flex translates the Lex specifications into a C source file called *lex.yy.c*, which is a lexical analyzer. Any lexical analyzer can be compiled using the following command.

```
[root@Zion testFlex]#cc lex.yy.y -lfl
```

This will compile the lexical analyzer, *lex.yy.c* (C source file), using any C compiler by linking it with the Flex library using the extension *-lfl*. After compilation, the output, by default, will write to *"a.out"* file. If you want to write the output to any one of the user defined file, then execute the following command to compile the lexer.

```
[root@Zion testFlex]#cc lex.yy.c -o program.output -lfl
```

In this case the output will write to the filename *program.output*. Now we can execute the resulting program to check out whether that will work out as expected. You may use the following commands to execute the same.

```
[root@Zion testFlex]#./a.out
```

or

```
[root@Zion testFlex]#./program.output (i.e. /<output filename>)
```

PROGRAM 3.2

Consider another Lex program, Program 3.2, which scans (or get the input from the keyboard) the name and prints the message "Hi..... <Name>.....Good Morning" when the Enter key or new line is found.

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

%{

//************************************************************************
//FLEX program to print the name when an ENTER key is pressed
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//************************************************************************

#include<stdio.h>
char Name[10]; //Global Declaration of variables Name[10]

%}

%%

[\n] {  /* First rule is matched when a new line is found */
        /* Following message will be shown along with the given input name */
        printf ("\nHi...%s.....Good Morning...\n",Name);
     }
```

```
%%

main()
{
   char opt;
   do
   {
      printf("\n\nWhat is your name:");
      scanf("%s",Name);//scanning your name
      yylex(); //calling the rules section function
      prinf("\nPress any key to continue(Y/y):");
      scanf("%c", &opt);
   }while ((opt == 'Y')||(opt== 'y'));
}
```

The definition section contains global declarations, including the header files and usual commands. Here there is a global declaration for storing the username.

The second part contains the Lex specification subroutine. You may note that this subroutine is using the variable which has been globally declared in the definition section. The Lex rule in this section will be matched when a new line is found.

In the third section, the user defined function section, we have normal C *main()* function which scans your name and prints the message whenever an Enter key is pressed. This procedure is repeated until we input the option as *N* (for No).

PROGRAM 3.3

Program 3.3, using the user defined function, prints the message *Hi….. <<your name>> Good Morning….* whenever a new line or Enter key is pressed.

```
%{

//*************************************************************************
//FLEX program to print the name, whenever an ENTER
// key is pressed (using the functions)
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************

void disply(char*); /*Function Declaration */

%}

%%
```

```
[\n] {
        char name[20];     /* Variable is declared */
        printf("Enter your name =");
        scanf("%s",name);/* Getting the input */
        disply(&name[0]);/* Function calling */
        return;      /* returning to the called function, which is main
        here */
    }
%%
/* Function definition to display the message */
void disply(char *in)
{
  printf("\nHi....%s Good Morning......\n",in);
}
main()
{
  printf("\n\nPress <<ENTER>> key to show the message");
  yylex();  /*  Calling  the  function  yylex(),  to  execute  the  rules
  section */
}
```

Note that this program is almost similar to Program 3.2, except that this is using the C function "disply" to print the message. Moreover, the character variable *Name* is not declared globally.

In the definition section we have a function declaration (i.e. disply). This function returns nothing (void) and gets a parameter of type char *. The definition of this function is in the user defined function section.

The second section contains rules. The pattern section of the rules contains the regular expression to denote the new line and its action part contains the variable declaration and other C statements. At the end, the function *disply()* is called by passing the parameter.

Note that the open curly bracket ("{") in the action part of any rule and the pattern should be in the same line with a single white space.
For example:

```
1. [\n]  {
              Printf("\n input is a newline");
         }
2. [\n]
         {
              Printf("\n input is a newline");
         }
```

Example 1 is the correct syntax while Example 2 is not.

Also note that in Lex program all the statements should begin at the extreme left column of the file (because Flex is left recursive tool). Otherwise, the Flex tool may not be able to generate the lexical analyzer properly, and thus it will be an error.

In the last user defined section, we can see the function *disply()* defined and the *main()* function.

PROGRAM 3.4

The following Flex program (Program 3.4) will check whether the given word is a vowel or not and print it.

```
---------------------------------------------------------------------------
%{

//*************************************************************************
//FLEX program to check whether the given word is
// vowels or NOT, using functions
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************

void display(int);

%}

%%

[a|e|i|o|u][a-zA-Z]+ { /* First Rule where the patterns are matched against its
vowels*/
   int flag=1;          /* Initializing the variables */
   display(flag);       /* Function Calling */
   return;              /* Return to where it is called */
   }

.+ {         /* Second rule, where the patterns
                are matched if any thing other than the above rule */
     int flag=0;
     display(flag);
     return;
}

%%

/* Defining the function 'display' to the apt message */
void display(int flag)
{
  if(flag==0)
       printf("\nThe given word is NOT a vowel\n");
  else
     printf("\nThe given word is a vowel\n");
```

```
}
main()
{
  printf("\nEnter a word to check whether it is vowel or NOT\n");
  yylex(); /* Calling the function to execute the rules in the rules section */
}
```

This program will check whether the given word starts with any one of the *a, e, i, o, u* characters. If so, it will print that the given word is a vowel, otherwise not. The first section of this program contains the usual C commands and a function declaration. One parameter is passed to the function *display()* of type int.

In the second section we have two rules. The rules stated first have the highest priority and will be executed first. If the pattern is not matching with the given input then it will execute second rules, and so forth. In this program, the pattern of the first rule checks whether the first letter starts with any one of the characters *a, e, i, o, u* and followed by any character of small case, a–z or a capital case, A–Z. If it matches, the variable *flag* is set to one and it is passed to the function *display*. The action part of the second rule will be executed only when it matches its pattern (.+), i.e. any thing other than what is said in the first pattern will match the second. The variable *flag* is set to zero if the given word is not a vowel.

In the third section, two functions are defined, i.e. the *display()* function that prints the message whether it is a vowel or not and the *main()* function from where the execution begins.

3.4 ANALYZING LEX PROGRAM WITH C PROGRAM

A Lex program is almost similar to a C program, except for the Flex syntaxes. If the reader closely analyzes the Flex programs discussed before, they might note the following points.

(a) In C program, anything declared outside the function is called *global declaration* (for example, #include, #define, variable declarations, function declarations, structure definition, etc.). In Flex, they are placed in the first definition section in between %{ or %} or outside any user defined C (auxiliary) functions in the last section.

(b) Any rule written in the second section, i.e. the rules section, will be converted by the Flex tool to a function named *yylex()*. So this is one of the important sections which need special attention when we discuss Flex tool or Lex program. Generally, we call this function *yylex()* from the *main()* function in the user defined section. Besides, the action part of any rule contains normal C code, which needs no conversion from Flex tool, because they are already in C code.

(c) The third and final section, i.e. the user defined section, contains normal user defined C functions. The necessary function that is to be defined in this section is *main()*, from where the execution begins. Generally any operations performed on normal C functions such as passing a parameter, pointers, returning a value can also be done in the Lex program.

3.5 yytext AND yyleng

When the generated lexical analyzer is executed, it analyzes its input looking for strings (tokens) that match any of its patterns. If it finds more than one pattern matching, the rule listed first in the Flex specification is chosen, and the corresponding action will be executed.

Once the match is determined, the text/string corresponding to the match (called the token) is made available in the global character pointer variable *yytext* (i.e. *yytext* is a globally declared variable by the Flex tool when it generates the lexical analyzer. By default its data type is character pointer). As soon as the lexical analyzer finds a token, it will be made available in the variable *yytext* for any further manipulations. Note that *yytext* will contain the recently found token. When another token is found, earlier one will be replaced with the recently matched token.

The user can define *yytext* in two different ways: either as a character pointer or as a character array. By default it is declared as a pointer. You can control the definition of *yytext* by including one of the special directives *%pointer* or *%array* in the first (definition) section of your Flex program. The advantage of using *%pointer* is substantially faster scanning and no buffer overflows when matching very large tokens (unless you run out of dynamic memory). The disadvantage is that *input()* function destroys the present contents of *yytext*, which can be a considerable porting headache when moved between different Lex version. The advantage of *%array* is that you can then modify *yytext* to your own will, and make calls to the *unput()* function (detailed discussion on *unput()* is in next section). We will discuss more on implementation of *yytext* in the next section with example.

%array defines *yytext* to be an array size of YYLMAX characters, which by default is a fairly large value. YYLMAX is a variable defined by the Flex tool when a lexical analyzer is generated. When the *yytext* is declared as array, by default it allocates the memory for the size YYLMAX. We can change the size by simply defining

```
#define YYLMAX <constant numbers>
```

to a different value in the first section of the Lex program. The concept of YYLMAX is used in Program 3.13.

But when we use *%pointer*, *yytext* grows dynamically to accommodate very large tokens (such as matching entire blocks of commands). Bear in mind that each time the scanner resizes *yytext*, it must also re-scan the entire token from the very beginning, so matching the tokens can prove to be a slow process. Moreover, existing Lex programs sometimes access *yytext* externally using declarations of the form

```
extern char yytext[];
```

That is, when a Lex program has to use or share other Flex program's *yytext* for some manipulations, we can execute the above statement to access the same.

The length of the string in *yytext* is copied to Flex variable *yyleng*. Whenever a new token is replaced with the old one, it will replace the content of *yyleng* accordingly. This is equivalent to execute the statement

```
int yyleng = strlen(yytext);
```

PROGRAM 3.5

Program 3.5 will check whether the given string is a word or a number. The word contains one or more (any) alphabetical characters from A to Z or (lower case) a to z letters. And a number contains one or more (any) digits from 0 to 9.

```
---------------------------------------------------------------------------
%{
//*************************************************************************
//FLEX program to check whether the given string is
// word or digit, using functions
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
void display(char[],int);
%}
%%
[a-zA-Z]+[\n] { /* First Rule to match a word with alphabet*/
  int flag=1;
  display(yytext,flag); /* Function is called by passing two
  parameters */
  return;
}
[0-9]+[\n] {  /* Second Rule to match a digit */
  int flag=0;
  display(yytext,flag); /* Function Calling */
  return;
}

.+{ /* Third rule is matched any thing other than the above rules */
  int flag=-1;
  display(yytext,flag); /* Function Calling */
  return;
}

%%
```

There are a number of special directives which can be included within an action. Directives, like keywords in C, are those words whose meaning has been already predefined in the Flex tool. Mainly, we have three directives in Flex.

1. "ECHO" copies *yytext* to the scanner's output. That is, whatever token we have recently found (or matched) will be copied to the output.
2. The directive BEGIN, followed by the name of the start symbol, places the scanner in the corresponding rules. Flex activates the rules by using the directive BEGIN and a start condition.
3. The directive REJECT directs the scanner to proceed into the "scanned best" rule to match the prefix of the input. That is, as soon as REJECT statement is executed in the action part, the last letter will be treated (or prefixed) from the recently matched token and will go ahead with the prefixed input for next best rule. To be more precise, Flex makes a DFA (Determinative Finite Automata) state machine out of your rules, so it only needs to keep track of the stacks it was in, and REJECT just jumps back to the last state. Program 3.6 will explain how REJECT statement works.

PROGRAM 3.6

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

%{

//*************************************************************************
//FLEX program to check REJECT statement
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************

%}

%%

[a-z]+ {    /* First Rule that matches small case alphabets*/
           printf("\nString contains only lower case letters =");
           ECHO; /* the content of yytext is displayed */
       }

[a-zA-Z]+ {  /* Second Rule that matches lower and upper case
                alphabets */
                printf("\nlts contains both lower & upper case letters
                =");
                ECHO;REJECT;   /* Last matched token is rejected */
             }
```

```
. {      /* Third Rule any one character other than the above rule*/
     printf("\nIts contains mixed letters =");
     ECHO;
}

%%

main()
{
  yylex();
}
```

In the rules section, we have three rules. The first rule's pattern will match combination of tokens with any number of lower case letters from a–z. The second will match a combination of tokens with any number of lower and upper case letters from a–z and A–Z. The third rule will match any unmatched character other than a–z and A–Z. Note that the third rule only matches one letter, while the other rules match one or more. Let us closely analyze the behaviour of Program 3.6 with the typical input *asDF*.

---------- output ----------

```
[root@localhost rejectTest]# ./a.out
asDF

Its contains both lower & upper case letters = asDF
Its contains both lower & upper case letters = asD
String contains only lower case letters = as
Its contains both lower & upper case letters = DF
Its contains both lower & upper case letters = D
String contains only lower case letters = D
Its contains both lower & upper case letters = F
String contains only lower case letters = F

[1]+ Stopped ./a.out
[root@localhost rejectTest]#
```

Flex always tries to match the longest possible string it can, so it matches *as* with the help of the first rule, but does not quit there since it is able to match *asD*. It does not stop there since it is also able to match *asDF* through the second rule. When it does so, it has run out of input, so it has to accept the string *asDF*. But it executes the action statements including REJECT, so it has to backtrack one state to where it matched with *asD*, which is prefixed (by one) input of *asDF*.

The second rule accepts string *asD* and executes the corresponding actions, including the REJECT again, and has to try with prefixed input *as* for the second rule as earlier. The third rule only matches any one of the character strings. It does not match with our prefixed input *as*, and has to try with other rules.

Now the first rule accepts *as* and runs the corresponding action statements, which do not include any "REJECT" or *return* to start over with a new input.

Flex goes back to the DFA states from where we have backtracked or prefixed, here it is *DF*. Flex matches the string through the second rule and execution of corresponding actions will REJECT again to prefix the input to *D*, which will be again matched by the second rule and go for the third rule to match the token. Then again it will go back to the translated prefix string *F* to match the second and third rules successively. Program 3.7 is a modified version of Program 3.6, that will explain all rules accepting (or matching) more than one letters.

PROGRAM 3.7

```
%{
//*************************************************************************
//FLEX program to check REJECT statement
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
%}
%%
[a-z]+  {   /* First Rule that matches small case alphabets*/
            printf("\nString contains only lower case letters =");
            ECHO; /* the content of yytext is displayed */
        }
[a-zA-Z]+ {   /* Second Rule that matches lower and upper case
                 alphabets */
                 printf("\nlts contains both lower & upper case letters =");
                 ECHO;REJECT;   /* Last matched token is rejected */
              }
.+{         /* Third Rule anything other than the above rule*/
       printf("\nlts contains mixed letters =");
       ECHO;
  }
%%
main()
{
  yylex();
}
```

In this program the third rule is matching with any number of letters that have failed to match by the first two rules. In this case, once the input *asDF* is matched through the second rule, REJECT statement will be executed when its action statements are executed. Then the matching will proceed for the second best rule with the prefixed input *asDF*, to find the last rule. The third rule matches and accepts the token *asD*. REJECT should be avoided at all costs when performance is important and is an expensive option. Also, note that unlike the other special actions, REJECT is a branch; the code (statements) immediately following it in the action will not be executed.

3.7 START CONDITION

Start conditions are declared in the definition (first) section of the Flex program using unintended lines beginning with either *%s* or *%x*, followed by a list of names called *start symbols*. If a start condition is declared with *%s*, then it is called an *inclusive start condition*, and if it is declared with *%x*, then it is called an *exclusive start condition*.

A start condition rule is activated using the directive BEGIN. Until the next BEGIN action is executed, rules with the given start conditions will be active and those with other conditions will be inactive.

If the start condition is declared as inclusive, then all rules without any start condition and rules with corresponding start condition will be active. If it is exclusive, then only rule(s) that is/are qualified with the start condition will be active. Programs 3.8 and 3.9 show how an inclusive start condition differs from an exclusive one.

BEGIN(<start condition>) will activate the corresponding declared rule(s) for the given start condition. BEGIN(0) returns to the original (initial) state where only the rules without start conditions are active. This state can also be referred to as the start condition INITIAL, therefore BEGIN(INITIAL) is equivalent to BEGIN(0).

PROGRAM 3.8

```
---------------------------------------------------------------------------
%{
//*************************************************************************
//FLEX program that implements inclusive start condition
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
%}
%s SM SMBG
%%
# BEGIN(SM);      /* First Rule that matches '#' */
## BEGIN(SMBG);   /* Second Rule that matches '##' */
```

```
[0-9]+  {     /* Third Rule that matches any digit */
          printf("\nIts a digit");
        }

<SMBG>[A-Z]+ {  /* Fourth Rule that will get activated when SMBG
                   begins and match wih A-Z */
                   printf("\nGiven string contains big letter(s)");
                }

<SM>.       {  /* Fifth rule */
          printf("\nExiting from # start condition");
          BEGIN(INITIAL);   /* Invoking initial start conditions */
  }

<SM, SMBG>[a-z]+ {  /* Sixth Rule */
              printf("\nGiven string contains small letters(s)");
  }

<SMBG>.+ {     /* Seventh Rule */
          printf("\nExiting from ## start conditions");
  }

.+{     /* Eight Rule */
       printf("\nNO action to execute");
    }

%%

main()
{
  printf("\nEnter # when u r expecting digits and small case letter strings");
  printf("\nEnter ## when u r expecting only big and small case letter strings"):
  yylex();
}
```

Program 3.8 has two start conditions and they are declared inclusively. %s SM SMBG declares two start conditions, i.e. SM and SMBG. In the second section it has eight rules. The rules one, two, three and eight will be active as soon as the function *yylex()* is called, and also note that they are not attached to any start symbol. Others will be inactive whenever we input '#'. The '#' will call and execute BEGIN(SM) to activate all the rules having start condition SM. In this case the rules fifth and sixth will be activated along with the earlier ones. If we input any strings of characters with lower case alphabet, from a–z, then the sixth rule will be executed, and if we input any numerical digit from 0–9, then the third rule will be executed. Execution of BEGIN(INITIAL) will initialize all the earlier conditions. That is, only one, two, three, and eight conditions will be active, and all the other rules that start with start symbol SM will be disabled.

Whenever we input "##", BEGIN(SMBG) will be called and all the rules having start condition SMBG will be activated. The rules 4th, 6th and 7th will be activated along with the earlier ones. Note that the 6th rule is activated only when any one of its start condition (i.e. SM or SMBG) is activated. If we input any:

(i) Strings of directives with lower case alphabets from a–z, then the sixth rule will be executed.

(ii) Numerical digits from 0–9, then the 3rd rule will be executed.

(iii) Strings of characters with upper case alphabets from A–Z, then the fourth rule will be executed.

And if any other input other than the above, is given, then the seventh rule, followed by the 8th rule, will be executed. Execution of BEGIN(INTIAL) will initialize to earlier condition, where the rules one, two, three and eight are active. All other rules that start with start symbol SMBG will be disabled.

Program 3.9 shows how an exclusive start condition behaves.

PROGRAM 3.9

```
---------------------------------------------------------------------------
%{

//************************************************************************
//FLEX program that implements exclusive start condition
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//************************************************************************

%}

%x SM SMBG

%%

# BEGIN(SM);      /* First Rule that matches '#' */
## BEGIN(SMBG);   /* Second Rule that matches '##' */

[0-9]+  {           /* Third Rule that matches any digit */
              printf("\nIts a digit");
        }
<SMBG>[A-Z]+ {  /* Fourth Rule that will get activated when SMBG
                   begins and match wih A-Z */
                   printf("\nGiven string contains big letter(s)");
                }
<SM>.+      {  /* Fifth rule */
           printf("\nExiting from # start condition");
           BEGIN(INITIAL);   /* Invoking initial start conditions */
  }
```

```
<SM, SMBG>[a-z]+ {   /* Sixth Rule */
            printf("\nGiven string contains small letter(s)");
  }

<SMBG>.+   {      /* Seventh Rule */
            printf("\nExiting from ## start conditions");
  }

.+{     /* Eight Rule */
       printf("\nNO action to execute");
    }
%%
main()
{
    printf("\nEnter # when u r expecting only small case letter strings");
    printf("\nEnter ## when u r expecting only big and small case letter strings"):
  yylex();
}
```

Here the start symbols are declared exclusively using *%x* in the first section of the Lex program. The second section contains eight rules. The rules one, two, three and eight will be active whenever the function *yylex()* is called. And also note that they are not attached to any start symbol.

Whenever we input '#', BEGIN(SM) will be called and all rules starting with SM will be activated. That is, only the rules fifth and sixth will be activated and all other rules will be disabled. Any input string with lower case letters will match and show the corresponding message. Input strings other than lower case letters will cause to execute 5th rule. The action part of the fifth rule will execute BEGIN(INITIAL) to initialize to initial conditions, i.e. the rules one, two, three and eight are active.

Whenever we input "##", BEGIN(SMBG) will be called and all rules starting with SMBG will be made active. That is, only the rules fourth, sixth and seventh will be disabled. Any input string with upper or lower case letters will be matched and the corresponding messages will be displayed and letters in the input string other than upper or lower case letters will cause to execute the seventh rule. The action part of the seventh rule will execute BEGIN(INITIAL) to initialize to initial condition, i.e. the rules one, two, three and eight are active. Note that start conditions do not have their own name space; *%s* and *%x* declare the names in the same fashion as *#define* does.

3.8 SPECIAL FUNCTIONS

Special functions are those that are available in the Flex libraries that could be used as normal statements.

3.8.1 yymore()

The special function *yymore()* will output the *yytext*, when execution of the action part of any rule that invoked *yymore()* ends. For example, consider Program 3.10 which has two rules, one that matches the lower case letters and the other that matches upper case letters only.

PROGRAM 3.10

```
----------------------------------------------------------------------------
%{
//*************************************************************************
//FLEX program that checks the special function yymore()
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
%}
%%
[a-z]+  {  printf("\nIts a lower case letter =");  /* First rule */
           ECHO;printf("\nBeginning of the 1st yymore");
           yymore();printf("\nEnd of the 1st yymore\n");
        }
[A-Z]+  { printf("\n\nIts a upper case letter =");  /* Second rule */
        ECHO;printf("\nBeginning of the 2nd yymore");
        yymore();printf("\nEnd of the 2nd yymore\n");
        }
%%
main()
{
  yylex();
}
-------------------------------output-----------------------------
[root@localhost yymmoreTest]# ./a.out
good MORNING
Its a lower case letter = good
Beginning of the 1st yymore
End of the 1st yymore
Good
Its a upper case letter = MORNING
Beginning of the 2nd yymore
End of the 2nd yymore
MORNING
```

```
[1]+ Stopped ./a.out
[root@localhost yymmoreTest]#
```

For a given input *good MORNING*, the first rule matches the token *good* and echoed to the output. Subsequently, the *yymore()* function will be executed and the content of the *yytext* (i.e. presently active token *good*) is echoed when the execution of the corresponding first rule finishes.

The second rule matches the *MORNING* and the presence of *yymore()* will echo the *yytext* at the end of the execution of the second rule.

3.8.2 yyless()

yyless(n) returns all characters, except the first *n* characters of the current token, back to the input stream, where they will be re-scanned when the scanner looks for the next match. *yytext* and *yyleng* are adjusted appropriately (i.e. *yyleng* will now be equal to *n*). The concept of *yyless* is explained in Program 3.11. It has two rules. The first rule matches any token with the lower case letters and the second one matches any token with lower and upper case letters.

PROGRAM 3.11

```
%{
//*************************************************************************
//FLEX program that uses yyless()
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************
%}
%%
[a-z]+  {   /* First Rule */
        printf("\n\nThe word is =");ECHO;
        yyless(2); /* Reducing 2 character and pushing back the rest */
        printf("\nThe word after yyless =");ECHO;
        }
[a-zA-Z]+ {     /* Second Rule */
          printf("\n\nThe mixed word is =");ECHO;
        }
%%
main()
{
  yylex();
}
```

```
---------------------------------- output ----------------------------------
[root@localhost yymmoreTest]# ./a.out

Nice morning

The mixed word is = Nice

The word is = morning
The word after yyless = mo
The word is = rning
The word after yyless = rn
The word is = ing
The word after yyless = in
The word is = g
The word after yyless = g

[1]+ Stopped ./a.out
[root@localhost yymmoreTest]#
---------------------------------------------------------------------------
```

The first word *Nice* is matched by the second rule and the following action statements are printed. And the given input *morning* is matched by the first rule and the following action statements are executed, including *yyless(2)*. It will return all the characters in the current token (i.e. morning) back to the input stream, except the first two characters where they will be re-scanned. Here *rning* is being re-scanned and the process is continued until it runs out of input. An argument of 0 to *yyless* will cause the entire current input string to be scanned again. This will cause an endless loop since the input is scanned again and again unless and until we change the scanner's action statements.

3.8.3 unput()

The function *unput(a)* puts or returns the character *a* back into the input stream and it will be the next character to be scanned. Program 3.12 will explain the concept of *unput()*.

PROGRAM 3.12

```
---------------------------------------------------------------------------
%{
//*************************************************************************
//FLEX program to check the function of unput(a),
// which returns the character a back to the input stream
//
//The lexer generated using the flex-2.5.4a tool in RedHat Linux EL
//*************************************************************************

%}
```