

BITS Pilani, Hyderabad Campus
Department of Computer Science and Information Systems
Second Semester, 2024-25
CS F363 Compiler Construction
Lab-4: Introduction to Flex/Lex tool

1 Objectives

The objectives of this lab sheet are given below.

1. To familiarize students with the basics of the Flex tool, its structure, and its application in generating lexical analyzers.
2. To understand the three main sections of a Flex program: definition, rules, and user code, and their roles in building a lexer.
3. To understand and apply regular expressions for token definitions and pattern matching.
4. To learn how to compile and execute Flex programs, analyze the generated C files, and observe the behavior of the lexer.
5. To demonstrate the use of Flex-specific variables such as yytext, yyleng, yyin, and yyout and understand their roles in input and output handling.

2 Introduction

Lex/Flex is a widely used tool that automates the process of creating lexical analyzers, which are crucial components in the development of compilers and interpreters. By default, Flex is included in most standard Linux operating systems. The primary purpose of Flex is to generate a lexer, also known as a scanner, which reads input streams of text and breaks them into a sequence of tokens based on predefined rules. These tokens serve as the building blocks for further stages of language processing, such as parsing.

Essentially, a lexer transforms the raw input text into meaningful units called tokens. Tokens are sequences of characters that represent identifiable elements of a language, such as keywords, identifiers, operators, or numbers. Flex facilitates this by allowing developers to define a set of rules, expressed as regular expressions (RE), that describe valid tokens. Flex then processes these rules and produces a C program, known as the lexer or scanner, which performs the tokenization task.

Regular expressions (RE) form the backbone of the token definitions in Flex. They are powerful constructs for pattern matching, capable of representing complex tokenization rules in a concise manner. Flex compiles these regular expressions into efficient code that scans the input text and identifies tokens according to the specified patterns.

A typical workflow with Flex involves writing a Flex specification file, compiling it to generate the C source code for the lexer, and then further compiling this C code to create an executable lexer. This lexer can process input streams, identify tokens, and execute corresponding actions, making it a vital tool for compiler construction, text processing, and many other applications in programming.

The figure below illustrates the input and output of a lexer:

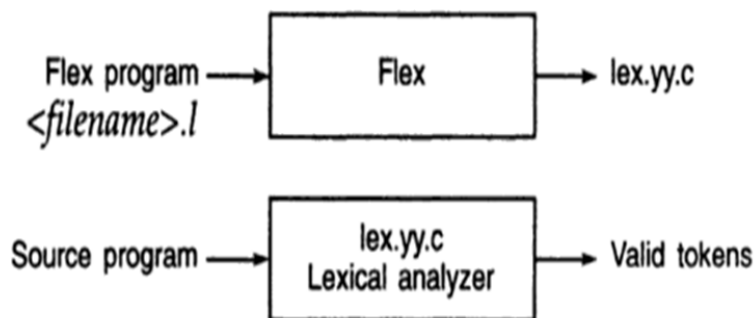


Figure 1: Input and output to a Lexer

By understanding the workings of Flex and its specification structure, developers can create robust and efficient lexical analyzers tailored to their specific requirements. This lab aims to provide a comprehensive introduction to the Flex tool, guiding students through its key features, usage, and practical applications in the context of tokenization and lexical analysis.

3 Structure of a Flex Program

Any Flex program consists of three sections separated by a line.

Definition Section

%%

Rules Section

%%

User Code (Auxillary) Section

3.1 Definition section

Three things can go in the definitions section:

- **C code:** Any indented code between `%{` and `%}` is copied to the C file. This is typically used to define file variables and prototype routines defined in the code segment.

```
%{
#include<stdio.h>
int count = 0; // Global variable
void displayCount();
%}
```

- **Flex definitions** A definition is very much like a `#define` C directive. For example

```
letter [a-zA-Z]
digit [0-9]
punct [!?]
nonblank [^\t]
```

These definitions can be used in the rules section: one could start a rule

```
{letter} {Some actions}
```

- **State definitions:** If a rule depends on context, it's possible to introduce states and incorporate those into the rules.

A state definition looks like `%s STATE_NAME`, and a state `INITIAL` is already given by default.

We will get into details of this in the next lab sheet.

3.2 Rules section

The rules section is the core of a Flex program, where patterns and their corresponding actions are defined. Each rule in this section consists of a pattern, written as a regular expression, and an associated action in the C code. When the (longest) prefix of the input matches a pattern, the corresponding action is executed. The rules section is enclosed between two sets of double percentage signs (`%% ... %%`).

3.2.1 Structure of a Rule

Each rule has the following format:

```
pattern { action }
```

- **Pattern:** A regular expression that defines the token to be recognized.
- **Action:** A block of C code that is executed when the pattern is matched. If the action is more than a single command, it should be enclosed in braces.

Here is an example of some typical rules:

```
[a-zA-Z]+ { printf("Word detected: %s\n", yytext); }
[0-9]+    { printf("Number detected: %s\n", yytext); }
\n        { printf("Newline detected\n"); }
.         { printf("Unknown character: %s\n", yytext); }
```

In this example:

- The first rule matches words made up of letters and prints them.
- The second rule matches sequences of digits and prints them.
- The third rule matches newlines and indicates when they occur.
- The last rule catches any character not matched by previous rules and labels it as unknown.

3.2.2 Important Notes on Patterns and Actions

- Patterns can include Flex definitions from the definitions section to simplify and modularize rules.
- Actions can modify variables, call functions, or perform any task C language supports.
- When no action is provided for a rule, the default action is to discard the matched text and continue scanning.

3.2.3 Conflict Resolution

Flex resolves conflicts using the following principles when multiple rules match the same input.

- **Longest Match:** If two or more rules match prefixes of the input, the rule that matches the longest sequence of characters is chosen, and the corresponding characters are taken as a valid lexeme.
For example, the pattern `[a-z]+` will match `"hello"` rather than stopping at `"h"` (see Example 3 for more understanding).
- **Rule Order:** If two matches are of the same length, the first rule in the list is selected (see Example 4). Therefore, rules should be ordered carefully to ensure the desired behavior.

3.3 User code section

The User Code section is where additional logic, such as main functions or helper functions, is implemented. It can contain the program's main entry point and any auxiliary functions that support the lexer. If left empty, a default `main()` function is provided, which simply calls `yylex()`.

If `main()` function is included, then it should call `yylex()` where `yylex` is the scanner built from the rules.

```
int main()
{
    yylex();
    return 0;
}
```

In addition, this section may contain some user-defined functions. See the below example.

```
%{
#include<stdio.h>
void printMessage(const char* message);
%}

%%
[a-zA-Z]+ { printMessage("Word detected!"); }
[0-9]+ { printMessage("Number detected!"); }
\n { return 0; }
%%

//user defined main function
int main() {
printf("Enter input: ");
yylex();
return 0;
}

//user defined print function
void printMessage(const char* message)
{
printf("%s\n", message);
}

```

4 Compilation and Running the flex program

- Save the program with extension `.l`, for example `first.l`
- Next step is to compile a lex file using lex compiler

```
$ lex first.l
```

This will create a C file `lex.yy.c` (will contain code in C for the lexer)

- Now compile the `lex.yy.c` file as below.

```
$ cc lex.yy.c -ll
```

This will create an executable file `a.out`.

The option `-ll` is necessary as it links the Flex (or Lex) library to your program.

- Now, execute the lexer as below.

```
$ ./a.out
```

Example 1 The following program counts the number of vowels and consonants in a given input string.

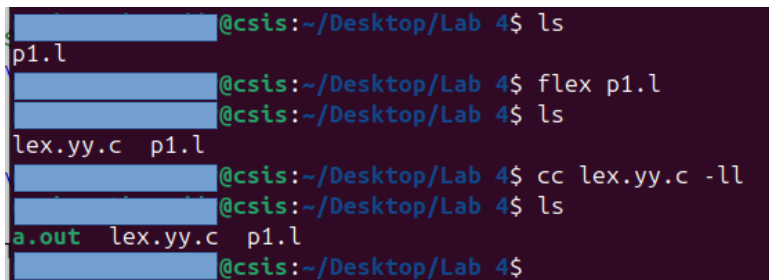
```
//C code section
%{
#include<stdio.h>
int vowels=0;
int cons=0;
%}

//flex defintions
vowels [aeiouAEIOU]
alphabets [a-zA-Z]
newline \n

%%
//Rules section
{vowels} {vowels++; }
{alphabets} {cons++;}
{newline} {return 0;}
%%

//User code section
int main(){
    printf("Enter the string:");
    yylex();
    printf("Output:\n");
    printf("No of vowels=%d \nNo of consonants=%d\n",vowels,cons);
return 0;
}
```

Try to compile the above program given in Example 1 (save the program with name p1.1) and observe the output. See the following figures; Figure 2 shows the compilation process and changes in the current directory, and Figure 3 shows the output of the program p1.1 for different inputs.



```
@csis:~/Desktop/Lab 4$ ls
p1.1
@csis:~/Desktop/Lab 4$ flex p1.1
@csis:~/Desktop/Lab 4$ ls
lex.yy.c  p1.1
@csis:~/Desktop/Lab 4$ cc lex.yy.c -ll
@csis:~/Desktop/Lab 4$ ls
a.out  lex.yy.c  p1.1
@csis:~/Desktop/Lab 4$
```

Figure 2: Compilation process

```

@csis:~/Desktop/Lab 4$ ./a.out
Enter the string:BITS Pilani Hyd Campus
Output:
No of vowels=6
No of consonants=13
@csis:~/Desktop/Lab 4$ ./a.out
Enter the string:BITSPilaniHydCampus
Output:
No of vowels=6
No of consonants=13
@csis:~/Desktop/Lab 4$

```

Figure 3: Output of p1.1 for two different inputs.

- Task 1** (a) Remove “{newline} {return 0;}” from p1.1 and observe the change in the programs behavior.
- (b) Change “{vowels} {vowels++; }” to “{vowels} {vowels++; return 0; }” in p1.1 and observe the output for the inputs in Figure 3.
- (c) Rewrite the p1.1 program without using flex definitions.

5 Some useful symbols to write the regular expressions

The regular expression symbols allowed in flex are shown in Table 1.

Metacharacter	Matches
.	Any character except newline
\n	Newline
*	Zero or more copies of preceding expression
+	One or more copies of preceding expression
?	Zero or one copy of 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

Table 1: Metacharacters and Their Matches

Example 2 Run the following program and check the output for the following inputs:

-
1. abbbbaababaaab
 2. ababaabbbbaaab
-

```
%{
#include<stdio.h>
%}

%%
(ab)+ printf("1");
ab+ printf("2");
a+b printf("3");
\n {printf("\n"); return 0;}
%%

int main(){
    printf("Enter the string:");
    yylex();
return 0;
}
```

Task 2 In the above program, change the operator + to * and observe the changes in the output for both inputs.

Example 3 Run the following program and check the output for the following inputs:

-
1. begin
 2. beginning
-

```
%{
#include <stdio.h>
%}

%%
begin printf("Compiler");
beginning printf("Compiler Design");
\n return 0;
%%

int main(){
    printf("Enter the string:");
    yylex();
return 0;
}
```

Example 4 Run the following program and check the output for the following inputs:

-
1. begin
 2. beginning
-

```

%{
#include <stdio.h>
%}

%%
begin printf("Compiler");
[a-z]+ printf("Compiler Design");
\n return 0;
%%
int main(){
    printf("Enter the string:");
    yylex();
return 0;
}

```

Task 3 In examples 3 and 4, change the order of the rules and observe the changes in the output for both inputs.

6 Some Flex variables

Some of the useful flex variables are given in the following table.

Name	Function
int yylex(void)	Call to invoke lexer, returns token
char *yytext	Pointer to matched string
yyleng	Length of matched string
yyval	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

Table 2: Lexer Functions and Definitions

Example 5 The following program illustrates the use of `yytext` and `yyleng`.

- `yytext` stores the lexeme that matches the rule.
- `yyleng` stores the length of the lexeme that matches the rule (in other words, it denotes the length of the lexeme stored in `yytext`).

```

%{
#include<stdio.h>
%}
%%
(ab)+ {printf("rule 1 lexeme: %s ",yytext); printf("length: %d\n",yyleng);}
ab+   {printf("rule 2 lexeme: %s ",yytext); printf("length: %d\n",yyleng);}
a+b   {printf("rule 3 lexeme: %s ",yytext); printf("length: %d\n",yyleng);}
\n    {printf("\n"); return 0;}
%%
int main(){
printf("Enter the string:");
yylex();
return 0;
}

```

```

@csis:~/Desktop/Lab 4$ ./a.out
Enter the string:abbbbaababaaab
rule 2 lexeme: abbb      length: 4
rule 3 lexeme: aab       length: 3
rule 1 lexeme: ab        length: 2
rule 3 lexeme: aaab      length: 4

@csis:~/Desktop/Lab 4$ ./a.out
Enter the string:ababaabbbbaaab
rule 1 lexeme: abab      length: 4
rule 3 lexeme: aab       length: 3
bbrule 3 lexeme: aaab    length: 4

raghunath-reddy@csis:~/Desktop/Lab 4$

```

Figure 4: Output of the program in Example 6.

Example 6 The following program illustrates the use of ECHO.

- ECHO prints the lexeme that matches the rule.

```

%{
#include<stdio.h>
%}
%%
(ab)+ {ECHO; printf(" length: %d\n",yyleng);}
ab+   {ECHO; printf(" length: %d\n",yyleng);}
a+b   {ECHO; printf(" length: %d\n",yyleng);}
\n    {printf("\n"); return 0;}
%%
int main(){
printf("Enter the string:");
yylex();
return 0;
}

```

Task 4 Run the code for the same input strings in Figure 4 and observe the output.

Note 1 Flex adds a rule “. ECHO; ” at the bottom of all rules. Due to this, if a character does not match any rule, it will be echoed onto the screen. You can observe the same in Figure 3 (for the first input, several spaces are echoed before “Output:.....”) and in Figure 4 (two b’s are echoed in the last line “bbrule 3 lexeme: aaab length: 4”).

Example 7 The following program illustrates how to read an input from a text file.

- To read input from a text file (say, `sample.txt`), we need to set the flex variable `yyin` to the file pointer of `sample.txt` or

```
yyin=fopen( "sample.txt", "r");
or
FILE *fp;
fp=fopen( "sample.txt", "r");
yyin=fp;
```

- Then call `yylex()`.

```
%{
# include <stdio.h>
%}
%%
(.|\n)* printf("%s", yytext);
%%
int main()
{
    yyin=fopen( "sample.txt", "r");
    yylex();
    return 0;
}
```

You can consider that `sample.txt` contains the following text:

```
Welcome to CS F363 Compiler Construction course
Lab sheet 4: Flex tool
BPHC Hyd 2024-2025 Second Sem
```

Task 5 (a) Run the code in Example 7 and observe the output.

- (b) Modify the code in Example 7 to count the number words (here, a word is collection of english alphabets) and integers in the `sample.txt` file.

Example 8 The following program illustrates how to write the output into a text file.

- To write the output into a text file (say, `output.txt`), we need to set the flex variable `yyout` to the file pointer of `output.txt` like `yyout=fopen("output.txt", "w");`
- Then call `yylex()`.

```
%{
# include <stdio.h>
%}
%%
[a-zA-Z]+ fprintf(yyout, "%s", yytext);
[ \t] {ECHO;}
. { }
%%
int main()
{
    yyin=fopen( "sample.txt", "r");
    yyout=fopen( "output.txt", "w");
    yylex();
return 0;
}
```

Task 6 Run the code in Example 8 and observe the output. Use the same `sample.txt` file given above.

7 Extra programs (Optional)

Example 9 Run the following program and observe the outputs for different inputs.

```
%{
#include<stdio.h>
%}

%%
aa    { printf("1"); }
b?a+b? { printf("2"); }
b?a*b? { printf("3"); }
.    { printf("4"); }
\n    {return 0; }
%%

int main(){
    printf("Enter the string:");
    yylex();
return 0;
}
```

Task 7 Write a FLEX program to count the number of lines, words, and characters from a C file.