BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
HYDERABAD CAMPUS
Compiler Construction (CS F363)
Lab sheet-3

LEXER

Lex/Flex is a tool that, by default, comes with the standard Linux Operating system to generate the lexical analyzer for any specification. Flex is a software tool for building a lexer.

A lexer takes input streams and tokenizes them into lexical tokens. Flex takes a set of rules for valid tokens and produces a C program called a lexer/scanner, that can identify these tokens. The set of rules or descriptions you give is called Flex specification.

The token description or rules that Flex uses are known as regular expressions(RE). Flex turns these RE into the lexer that scans the input text and identifies the tokens.

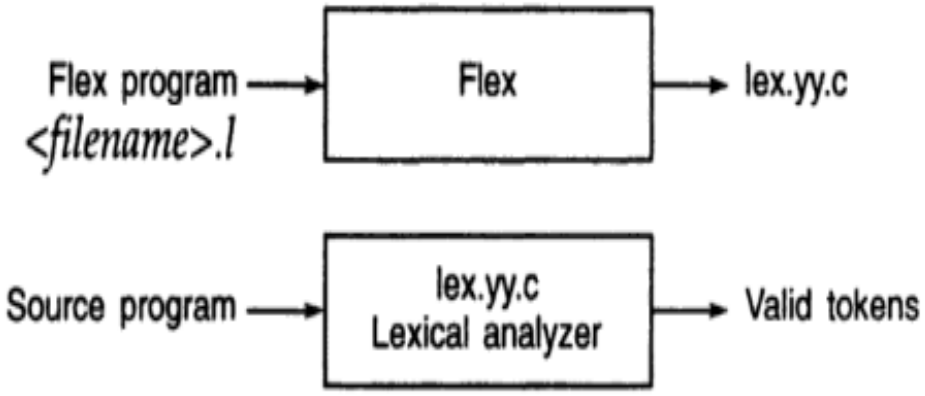
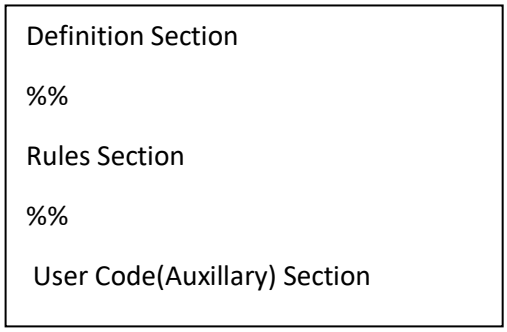


Figure 2: Input and output to a Lexer

Structure of a flex program

Any Flex program consists of three sections separated by a line with %%



In the example you just saw, all three sections are present:

Definitions: All code between %{ and %} is copied to the beginning of the resulting C file.

Rules: Several combinations of pattern and action: if the action is more than a single command, it needs to be in braces.

Code: This can be very elaborate, but the main ingredient is the call to yylex, the lexical analyzer. If the code segment is left out, a default main is used, which only calls yylex.

// try
%%
begin printf("Hello");
%%

This will work.

2.1.2 Definitions 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 for defining file variables and for prototypes of routines that are defined in the code segment.

definitions A definition is very much like a #define cpp 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}+ {...

state definitions: If a rule depends on context, it's possible to introduce states and incorporate those in the rules. A state definition looks like %s STATE, and by default a state INITIAL is already given. We will get into details of this in later labs.

2.1.3 Rules section

The rules section has several pattern-action pairs. The patterns are regular expressions; the actions are either a single C command or a sequence enclosed in braces.

The longer match is taken if more than one rule matches the input. If two matches are the same length, the earlier one in the list is taken.

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 preceeding 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

Table1: Pattern Matching primitives in flex.

2.1.4 User code section

If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty you will get the default main:

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

where yylex is the parser that is built from the rules.

Try this:

```
%{
#include <stdio.h>
%}
 letter  [a-zA-Z]
 digit [0-9]
%%

{letter}+ printf(“String of Alphabets \n”);
{digit}+    printf("Number");
%%
main()
{
yylex();
}
```

Save the program with some name, say 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 .c file as below.

~$ **cc lex.yy.c -ll**

This will create an executable file that lexer

Now execute the lexer as below.

~$ **./a.out**

Note:

1. The lab session will discuss other important content and practice problems.
2. This lab sheet will give a rough idea about the coverage but not the complete details.
3. Hence, attending the Lab session is important.