STRUCTURE OF LEX PROGRAM

A Lex program consists of three sections, separated by a line consisting of two percent signs (`%%`):

1. **Definition Section**
2. **Rules Section**
3. **Auxiliary Section**

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

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

Definition Section

The Definition Section contains user-defined Lex options used by the lexer. It creates an environment for the execution of the Lex program and can be empty. This section helps in two ways:

1. **Environment for the Lexer:**
   - Contains C statements such as global declarations and commands.
   - Enclosed by `%{` and `%}`.
   - Includes global declarations, commands, and tool configurations.
2. **Environment for Flex Tool:**
   - Provides declarations of simple name definitions to simplify scanner specifications.
   - Declares start conditions.
   - Helps Flex convert the Lex specifications correctly and efficiently to the lexical analyzer.

Rules Section

The Rules Section contains the patterns and actions that define the Lex specifications:

- **Patterns:**
  - Formed by regular expressions to match the largest possible string.
- **Actions:**
  - Enclosed in braces `{}`.
  - Contain normal C language statements.

  - When a pattern is matched, the corresponding action is invoked.
  - The lexer tries to match the largest possible string. If two rules match the same length, the lexer uses the first rule to invoke its corresponding action.

Auxiliary Section

This section contains user-defined C functions (subroutines), including the `main()` function from where execution begins. These functions are copied as-is to the lexical analyzer C file by FLEX.

Lex Variables

- `yyin`:
  - Type: `FILE*`
  - Points to the current input file.
- `yyout`:
  - Type: `FILE*`
  - Points to the output location.
- `yytext`:
  - Stores the text of the matched pattern in a variable.
- `yylen`:
  - Gives the length of the matched pattern.

Lex Functions

- `yywrap`:
  - Called when the end of the input file is encountered.
  - Can be used to parse multiple input files.
- `yylex(int n)`:
  - Can be used to push back all but the first `n` characters of the read token.
- `yymore`:
  - Keeps the token's lexeme in `yytext` when another pattern is matched.

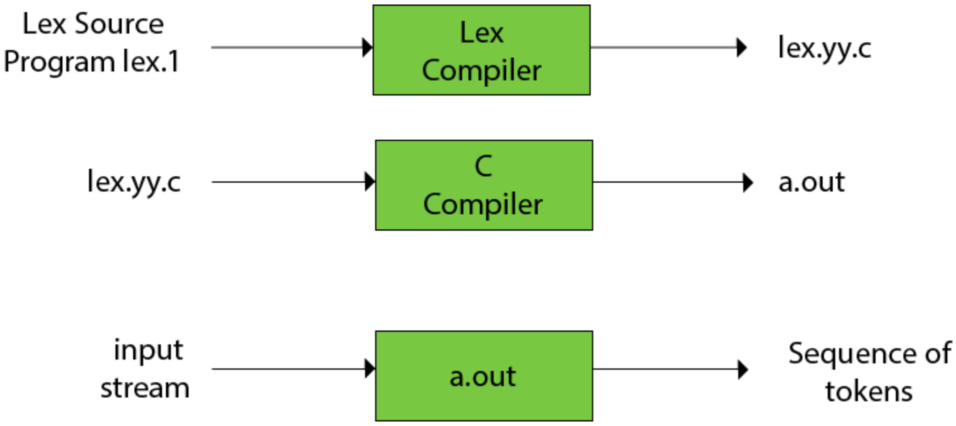
Lex Macros

- **`Letter`**: `[a-zA-Z]`
- **`Digit`**: `[0-9]`
- **`Identifier`**: `{letter}({letter}|{digit})`

INTRODUCTION

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



`yylex()`: The main Lex function that performs lexical analysis and matches patterns in the input.

`yytext`: A pointer to the matched text (a string) for the current pattern.

`yyleng`: The length of the matched text in `yytext`.

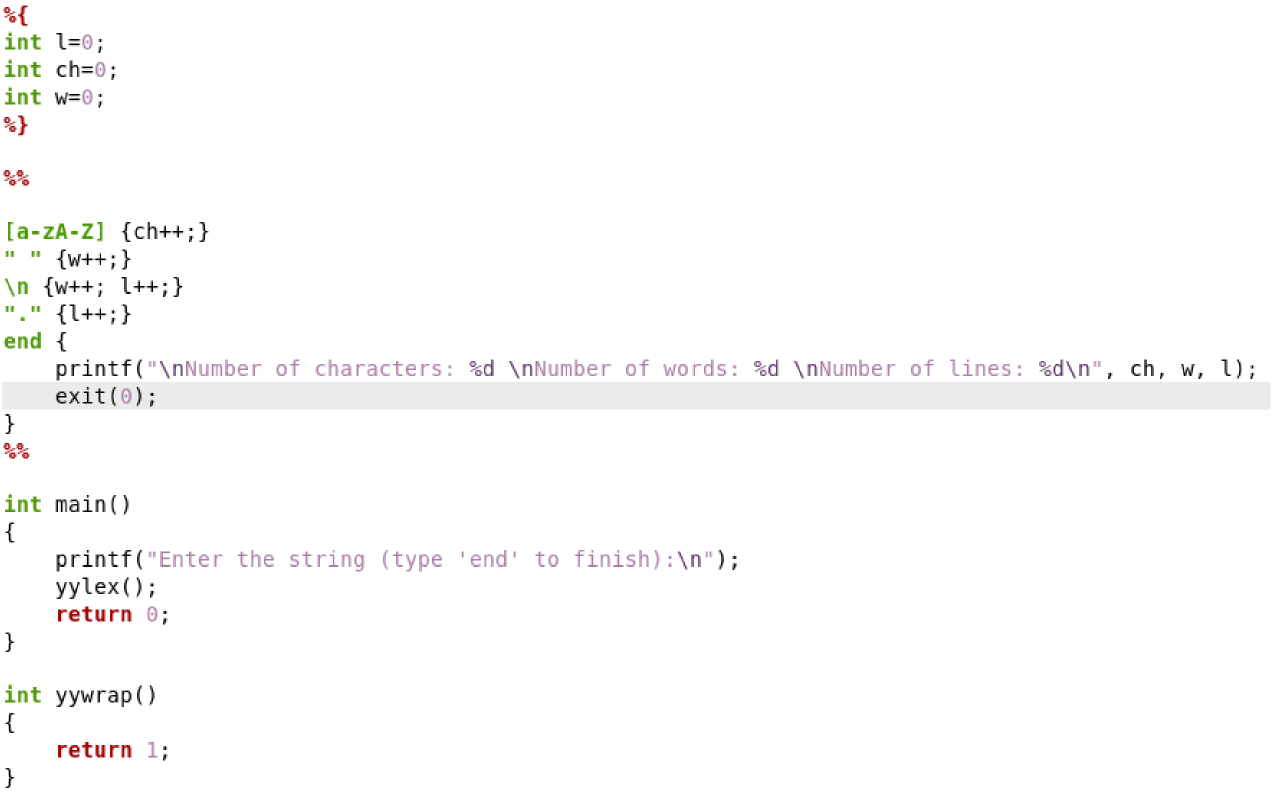
`yyin`: A file pointer that indicates the input stream (defaults to `stdin`).

`yyout`: A file pointer for output (defaults to `stdout`).

`yywrap()`: A function called when the end of input is reached; by default, returns 1 to indicate end of input.

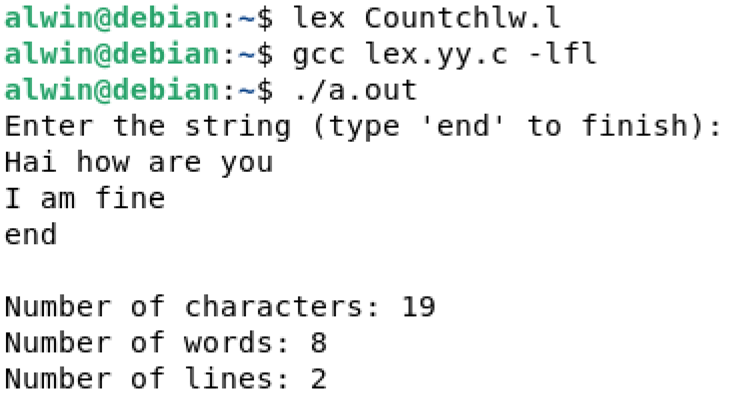
Experiment-1a: LEX program to count the number of lines, words and characters in an input and input from a file.

Countchlw.l



```
%{
int l=0;
int ch=0;
int w=0;
%}

%%

[a-zA-Z] {ch++;}
" " {w++;}
\n {w++; l++;}
"." {l++;}
end {
    printf("\nNumber of characters: %d \nNumber of words: %d \nNumber of lines: %d\n", ch, w, l);
    exit(0);
}
%%

int main()
{
    printf("Enter the string (type 'end' to finish):\n");
    yylex();
    return 0;
}

int yywrap()
{
    return 1;
}
```

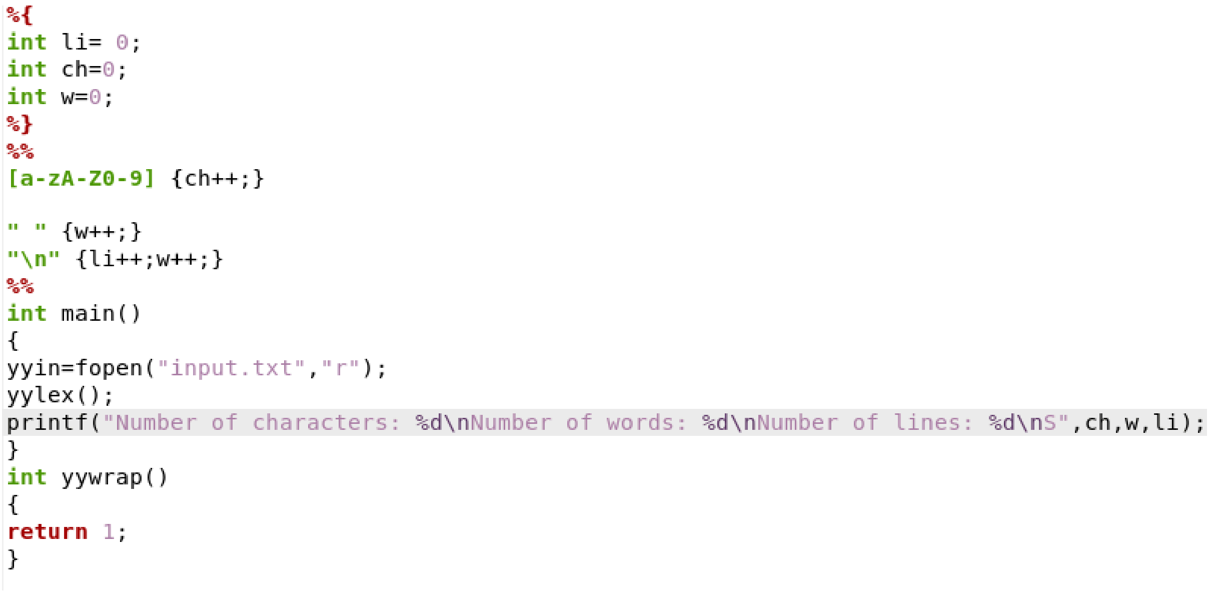
OUTPUT:



```
alwin@debian:~$ lex Countchlw.l
alwin@debian:~$ gcc lex.yy.c -lfl
alwin@debian:~$ ./a.out
Enter the string (type 'end' to finish):
Hai how are you
I am fine
end

Number of characters: 19
Number of words: 8
Number of lines: 2
```

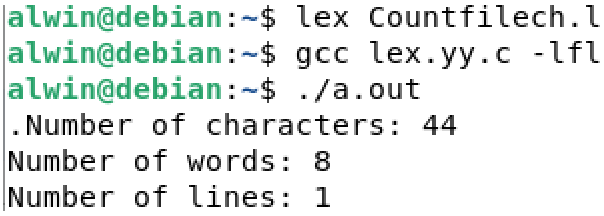
Experiment-1b: LEX program to count number of words, lines and characters from file

Countfilech.l



```
%{
int li= 0;
int ch=0;
int w=0;
%}
%%
[a-zA-Z0-9] {ch++;}

" " {w++;}
"\n" {li++;w++;}
%%
int main()
{
yyin=fopen("input.txt","r");
yylex();
printf("Number of characters: %d\nNumber of words: %d\nNumber of lines: %d\nS",ch,w,li);
}
int yywrap()
{
return 1;
}
```

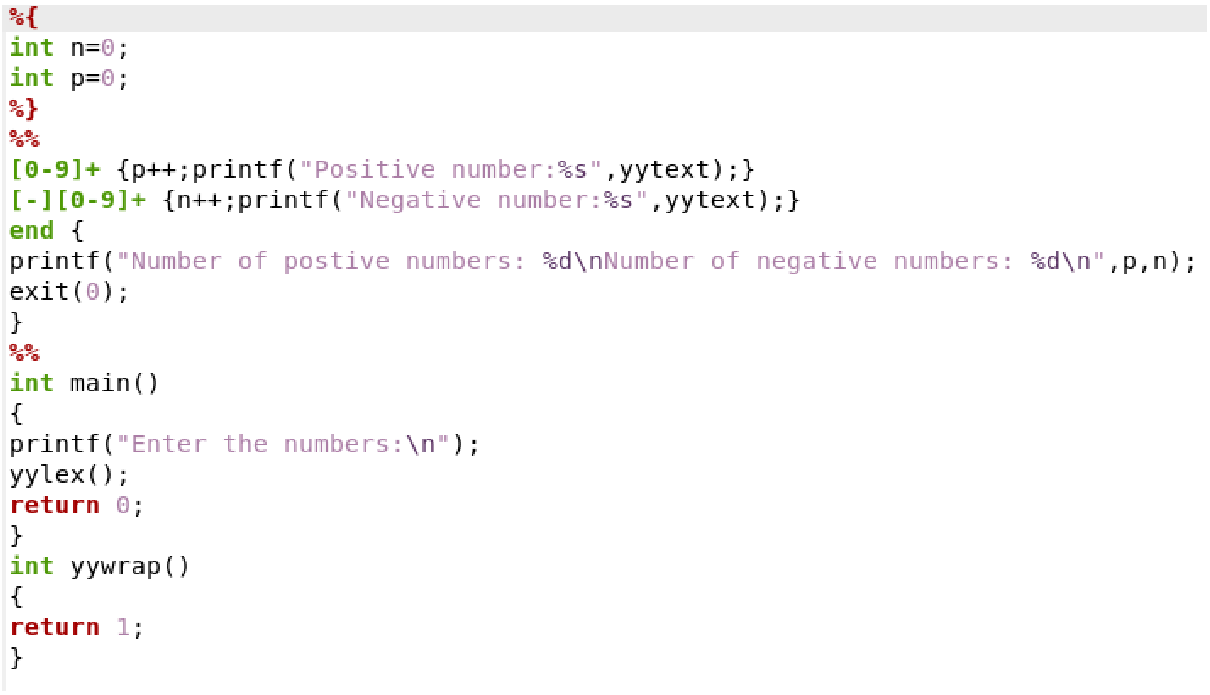
OUTPUT:



```
alwin@debian:~$ lex Countfilech.l
alwin@debian:~$ gcc lex.yy.c -lfl
alwin@debian:~$ ./a.out
.Number of characters: 44
Number of words: 8
Number of lines: 1
```

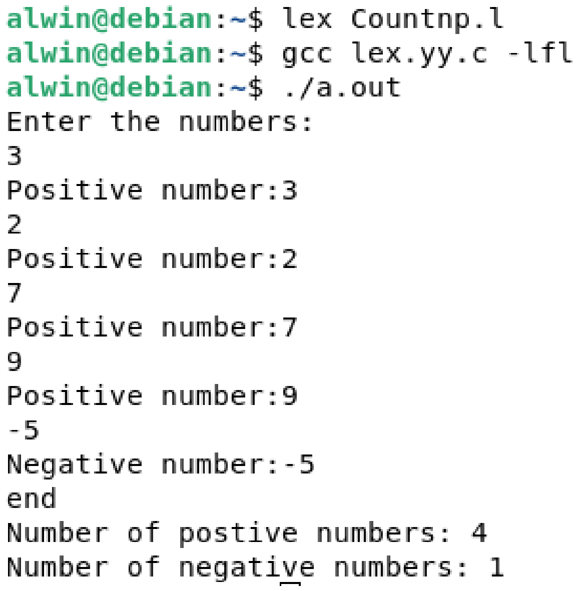
Experiment-2: LEX program to identify and Count Positive and Negative Numbers.

Countnp.l



```
%{
int n=0;
int p=0;
%}
%%
[0-9]+ {p++;printf("Positive number:%s",yytext);}
[-][0-9]+ {n++;printf("Negative number:%s",yytext);}
end {
printf("Number of postive numbers: %d\nNumber of negative numbers: %d\n",p,n);
exit(0);
}
%%
int main()
{
printf("Enter the numbers:\n");
yylex();
return 0;
}
int yywrap()
{
return 1;
}
```

OUTPUT:



```
alwin@debian:~$ lex Countnp.l
alwin@debian:~$ gcc lex.yy.c -lfl
alwin@debian:~$ ./a.out
Enter the numbers:
3
Positive number:3
2
Positive number:2
7
Positive number:7
9
Positive number:9
-5
Negative number:-5
end
Number of postive numbers: 4
Number of negative numbers: 1
```

Experiment-3: LEX program to count the number of vowels and consonants.

Countvc.l

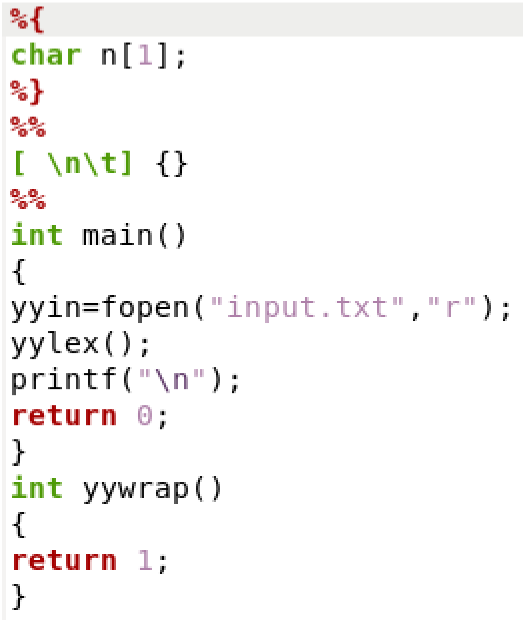
```
%{
int vow_count=0;
int const_count =0;
%}
%%
[aeiouAEIOU] {vow_count++;}
[a-zA-Z] {const_count++;}
"\n" {
printf("Vowels = %d \nConsonants = %d\n",vow_count, const_count);
exit(0);
}
%%
int main()
{
printf ("Enter the string of vowels and consonants:\n");
yylex();
return 0;
}
int yywrap()
{
return 0;
}
```

OUTPUT

```
alwin@debian:~$ lex Countvc.l
alwin@debian:~$ gcc lex.yy.c -lfl
alwin@debian:~$ ./a.out
Enter the string of vowels and consonants:
Computer Science
 Vowels = 6
Consonants = 9
```

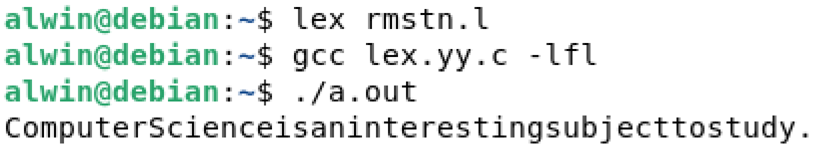
Experiment-4: LEX program to remove space, tab or newline.

rmstn.l



```
%{
char n[1];
%}
%%
[ \n\t] {}
%%
int main()
{
yyin=fopen("input.txt","r");
yylex();
printf("\n");
return 0;
}
int yywrap()
{
return 1;
}
```

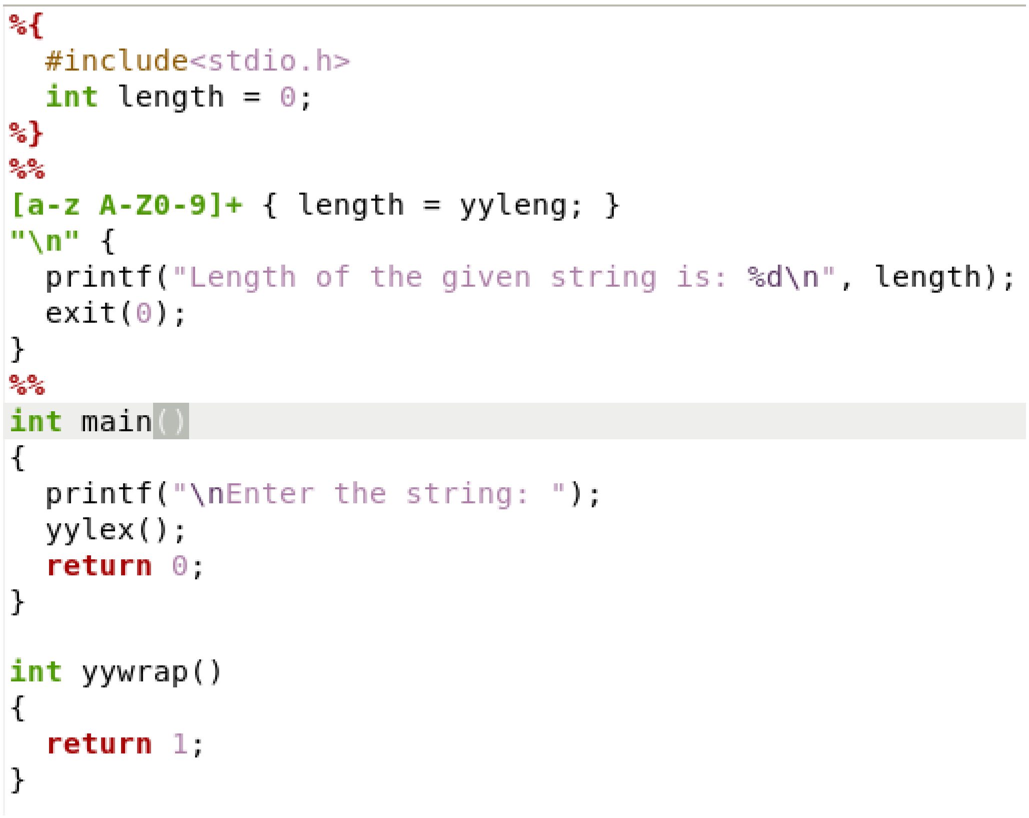
OUTPUT



```
alwin@debian:~$ lex rmstn.l
alwin@debian:~$ gcc lex.yy.c -lfl
alwin@debian:~$ ./a.out
ComputerScienceisaninterestingsubjecttostudy.
```

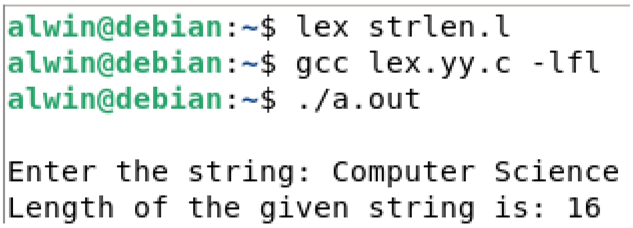
Experiment-5: LEX program to find the length of a string.

strlen.l



```
%{
  #include<stdio.h>
  int length = 0;
%}
%%
[a-z A-Z0-9]+ { length = yyleng; }
"\n" {
  printf("Length of the given string is: %d\n", length);
  exit(0);
}
%%
int main()
{
  printf("\nEnter the string: ");
  yylex();
  return 0;
}

int yywrap()
{
  return 1;
}
```

OUTPUT



```
alwin@debian:~$ lex strlen.l
alwin@debian:~$ gcc lex.yy.c -lfl
alwin@debian:~$ ./a.out

Enter the string: Computer Science
Length of the given string is: 16
```