**NAME : B.Mokshitha**

**CLASS : CSE-B**

**ROLL NO : 22171**

**COURSE : COMPILER DESIGN**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

***BASIC LEX PROGRAMS***

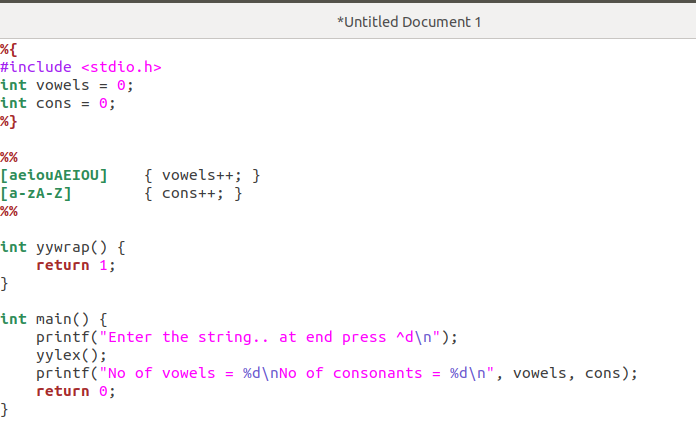
**AIM:**

Program to Identify Vowels and Consonants.

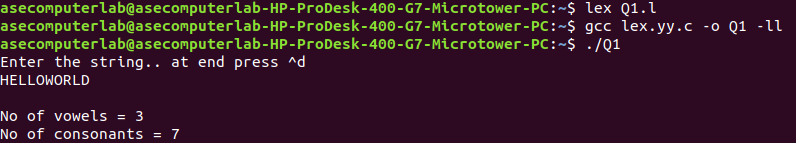
**ALGORITHM:**

* Start the Lex program and include necessary header files.
* Define regular expressions to match vowels, consonants, whitespace, and other characters.
* Implement Lex rules to classify characters based on the patterns.
* Use printf() to display the classification result for each input character.
* Call yylex() in main() and handle end-of-input using yywrap().

**CODE:**

****

**OUTPUT:**

****

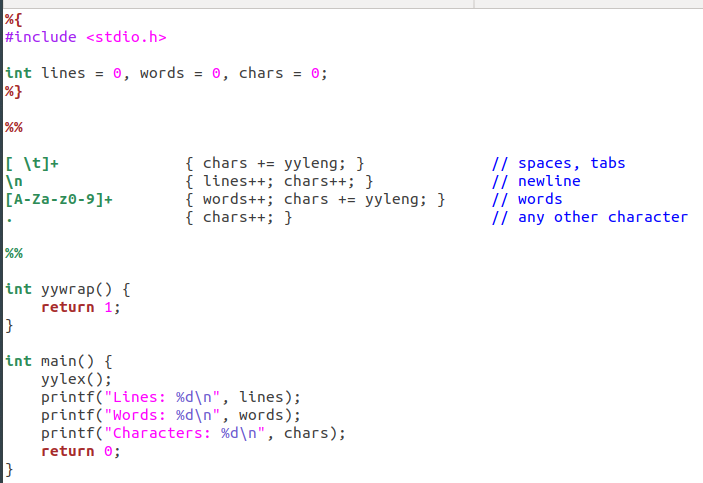
**AIM:**

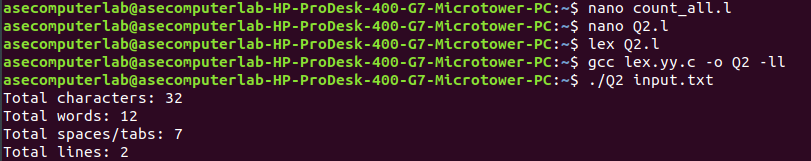
Program to Count Lines, Words, and Characters.

**ALGORITHM:**

* Start the Lex program and declare variables to count lines, words, and characters.
* Define regular expressions to match newlines, whitespace, and words.
* Use actions to increment counters based on the matched input (lines, words, or characters).
* Call yylex() in the main() function to start scanning the input.
* After scanning, print the total count of lines, words, and characters, and return from the program.

**CODE:**

****

**OUTPUT:**

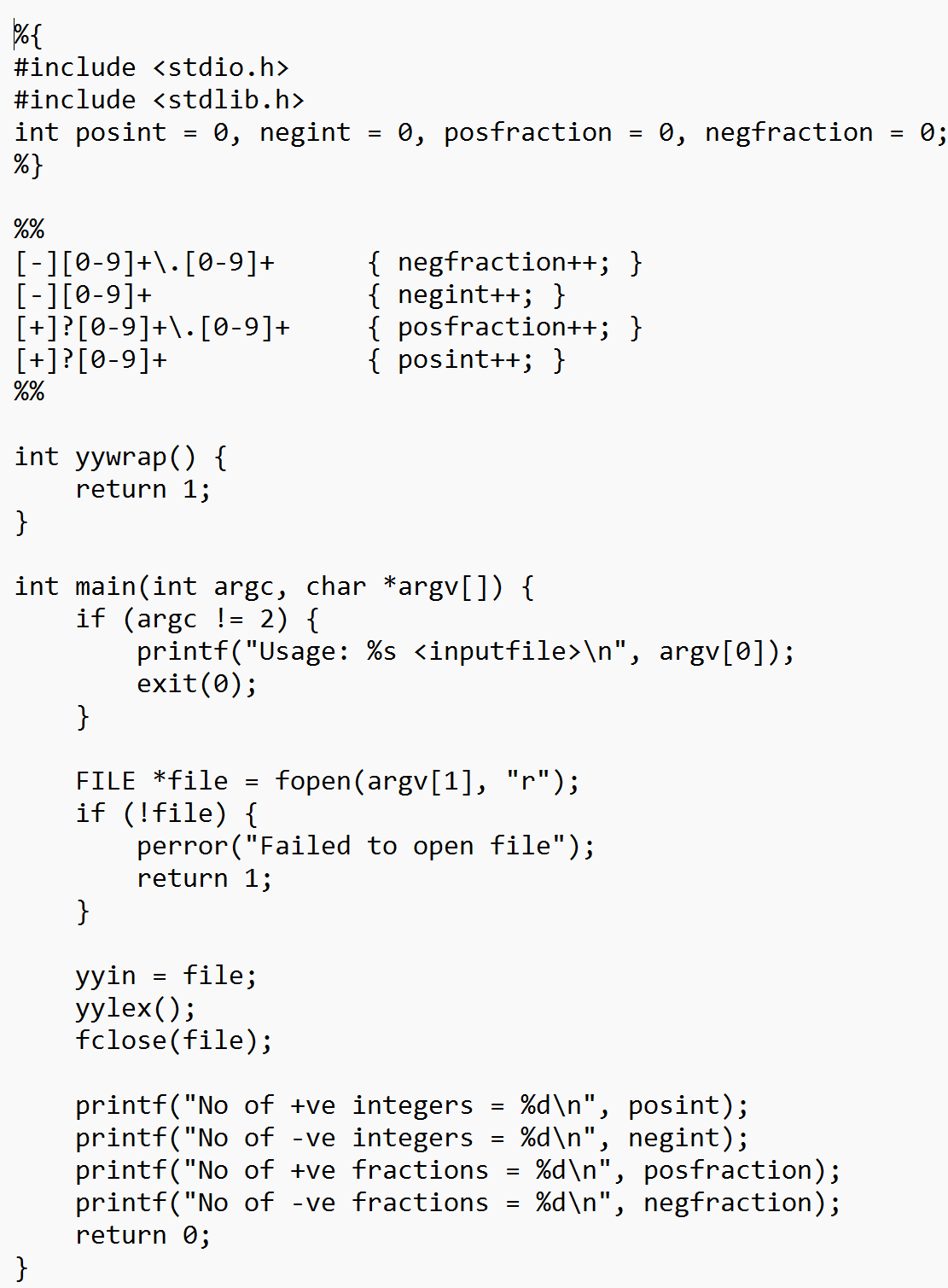
**AIM:**

To write a program that takes a list of numbers (both integers and fractions) as input and counts how many of them are positive and how many are negative.

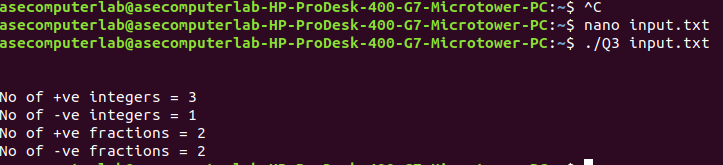
**ALGORITHM:**

* Start the Lex program and include the necessary header file.
* Initialize four counters:
* pos\_int\_count = 0
* neg\_int\_count = 0
* pos\_frac\_count = 0
* neg\_frac\_count = 0
* Take input of the total number of elements, n.
* Loop from 1 to n:  
  a. Accept the number as input (can be integer or fraction).  
  b. Check if the number is an integer or fraction.  
  - If integer:  
  - If number > 0, increment pos\_int\_count.  
  - Else if number < 0, increment neg\_int\_count.  
  - If fraction:  
  - If number > 0, increment pos\_frac\_count.  
  - Else if number < 0, increment neg\_frac\_count.
* After the loop ends, print all four counts.
* End the program.

**CODE:**

****

**OUTPUT:**

****

**AIM:**

Program to count the number of comment lines in a given C program and eliminate them, copying the cleaned program into a separate file.

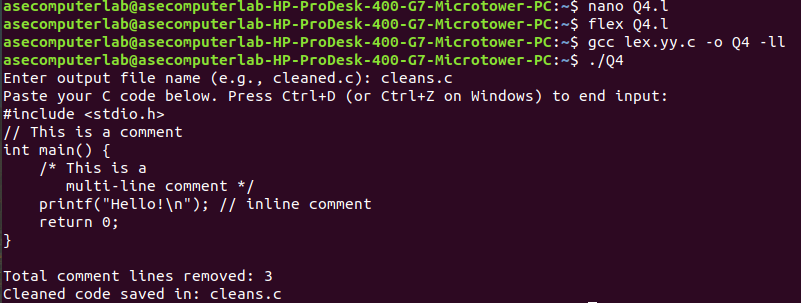
**ALGORITHM:**

* Start the Lex program and include required header files.
* Open the original C program file for reading.
* Open a new file for writing the cleaned program.
* Initialize a counter comment\_line\_count to 0.
* Read the C program file line by line:  
  a. For each line, check if it contains:
* Single-line comment: lines starting with //.
* Multi-line comment: lines between /\* and \*/.  
  b. If the line is a comment line (either fully or partially), increment comment\_line\_count.  
  c. If the line is not a comment line, write it to the new file.
* After processing all lines, close both files.
* Print the total number of comment lines found and removed.

**CODE:**

****

**OUTPUT:**

****

**RESULT:**

Therefore, basic lex programs executed successfully.

***LAB-1***

**AIM:**

To implement Lexical Analyzer Using Lex Tool.

**ALGORITHM:**

1.Open gedit text editor from accessories in applications.

2. Specify the header files to be included inside the declaration part (i.e. between %{ and %}).

3. Define the digits i.e. 0-9 and identifiers a-z and A-Z.

4. Using translation rule, we defined the regular expression for digit, keywords,identifier, operator and header file etc. if it is matched with the given input then store and display it in yytext.

5. Inside procedure main(),use yyin() to point the current file being passed by the lexer.

6. Those specification of a lexical analyzer is prepared by creating a program lexp.l in the LEX language.

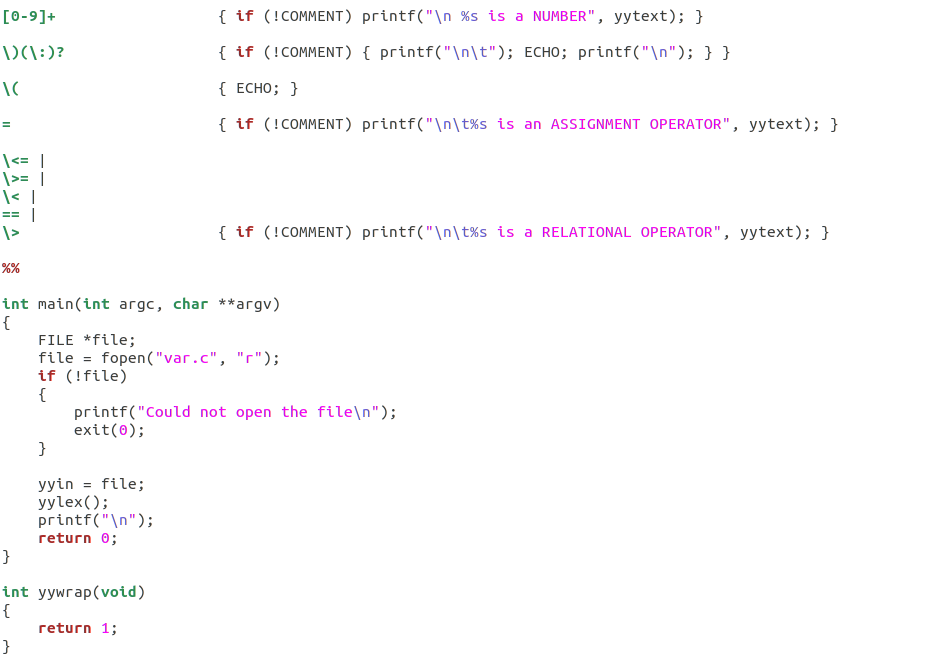
7. The Lexp.l program is run through the LEX compiler to produce an equivalent code in C language named Lex.yy.c .

8. The program lex.yy.c consists of a table constructed from the Regular Expressions of Lexp.l, together with standard routines that uses the table to recognize lexemes.

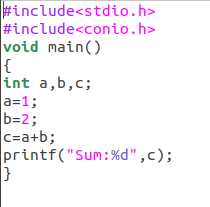
9. Finally, lex.yy.c program is run through the C Compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

**CODE:**

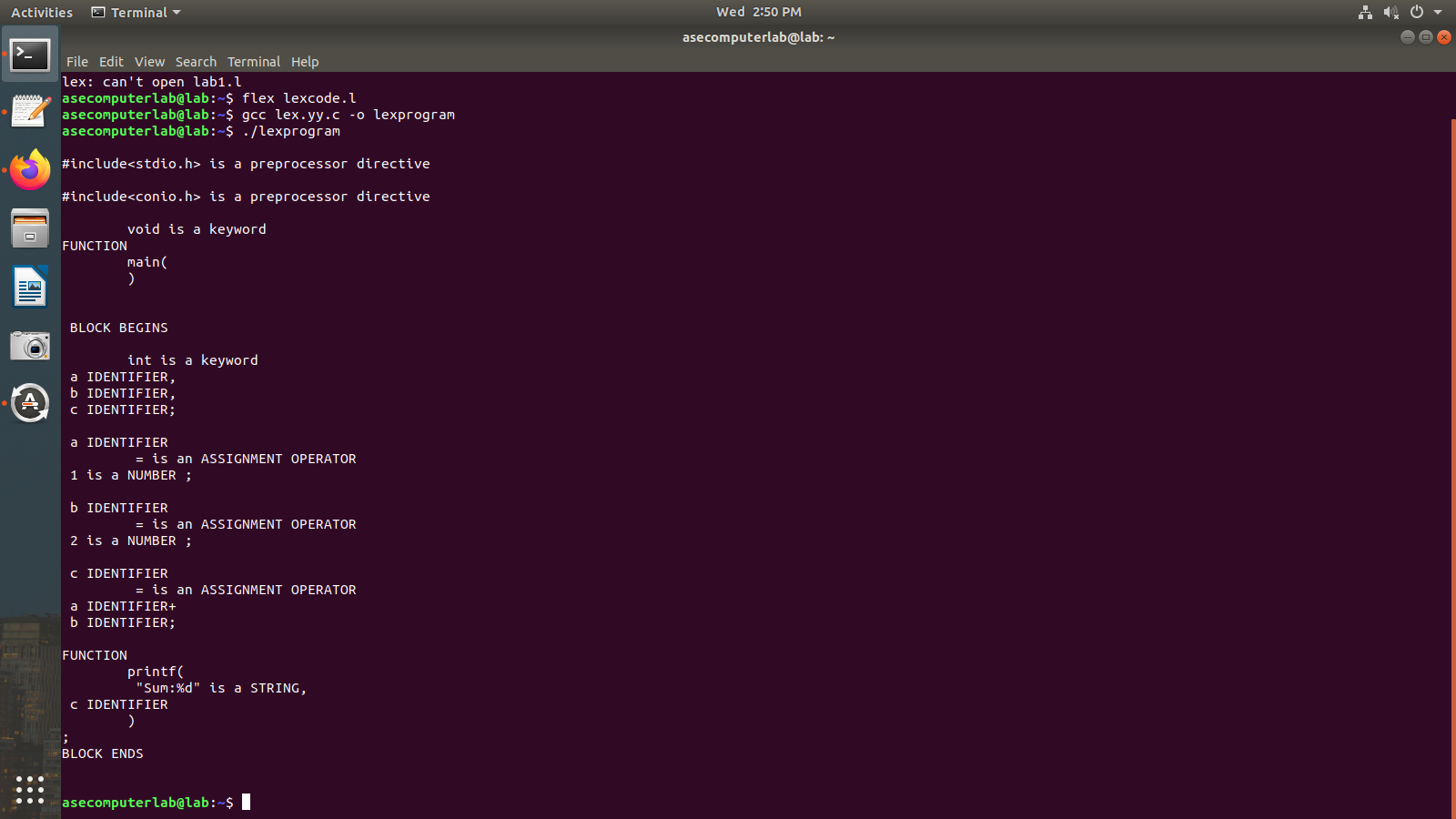
**Var.l**

****

**Var.c**

****

**OUTPUT:**

****

**RESULT:**

The code has been executed and output displayed successfully.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

***LAB-2***

**AIM:**

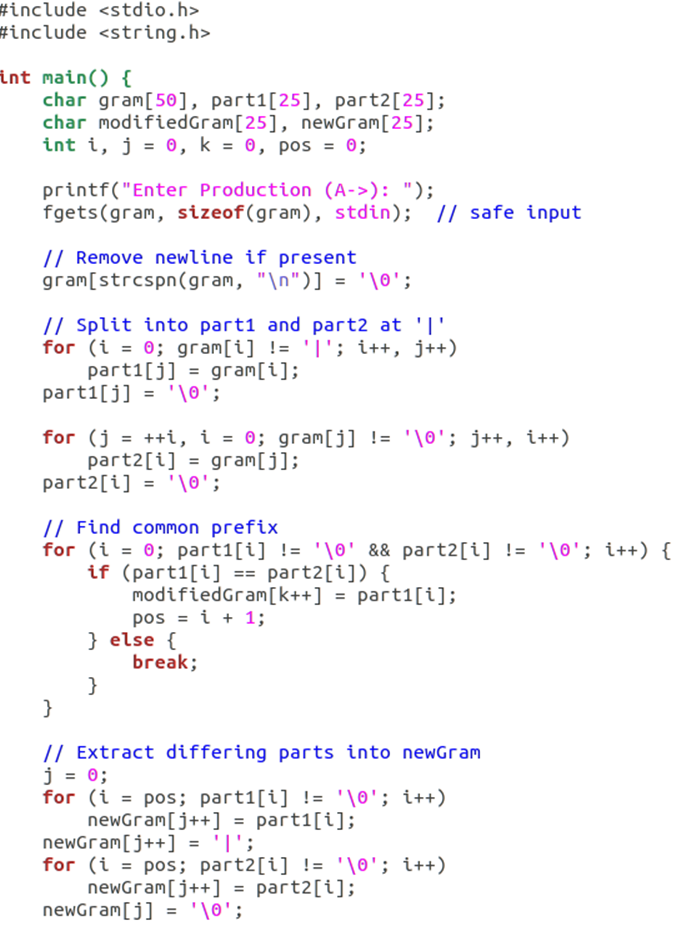
To implement eliminate left recursion and left factoring from the given grammar using C program.

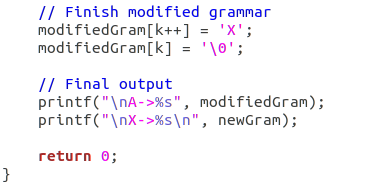
**ALGORITHM:**

**Left Factoring:**

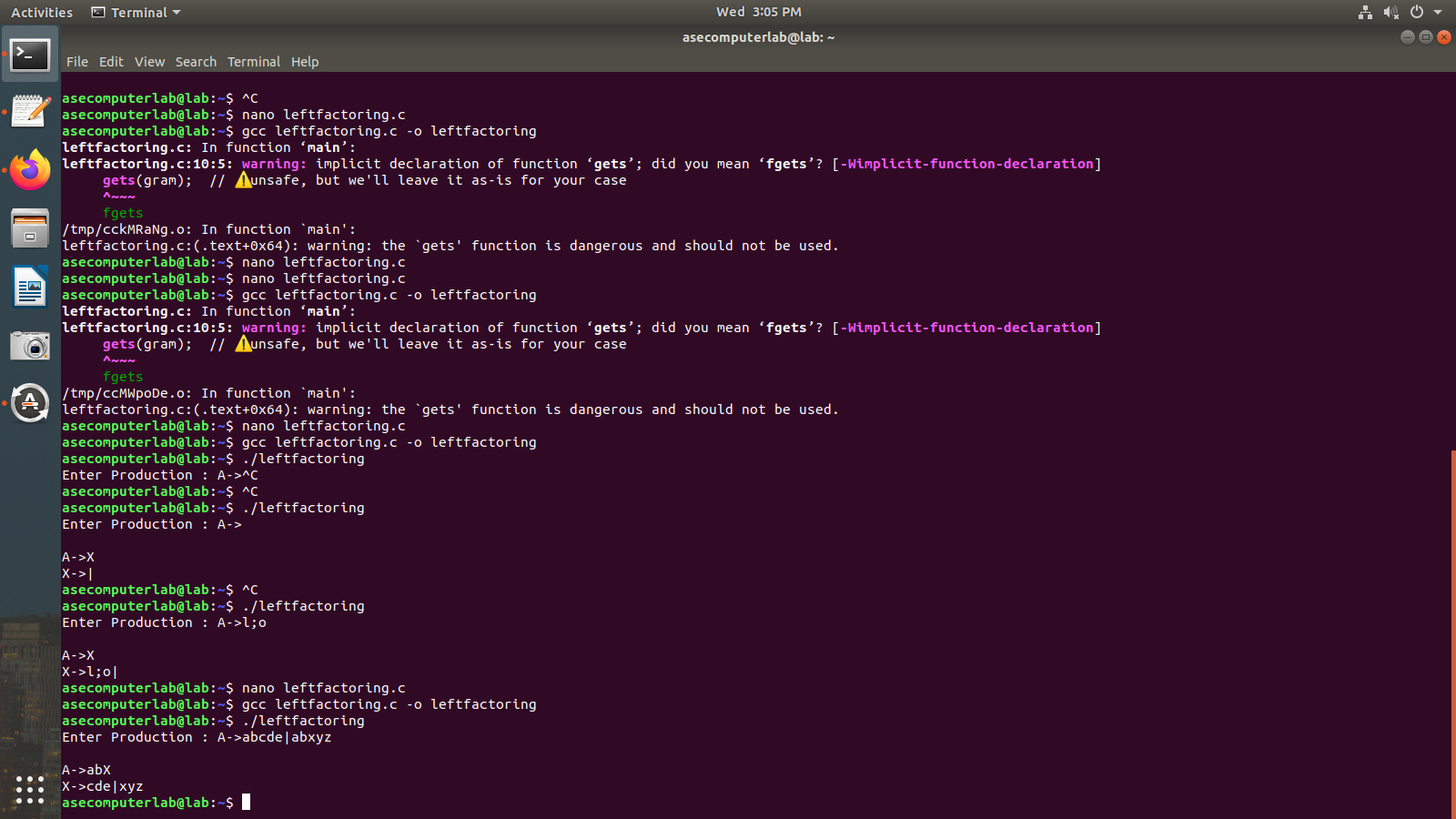
* Start the processes by getting the grammar and assigning it to the appropriate variables.
* Find the common terminal and non-terminal elements and assign them in a separate grammar.
* Display the new and modified grammar.

**CODE:**

****

****

**OUTPUT:**

****

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

***LAB-3***

**AIM:**

Left recursion

**Algorithm:**

1. Start the processes by getting the grammar and assigning it to the appropriate

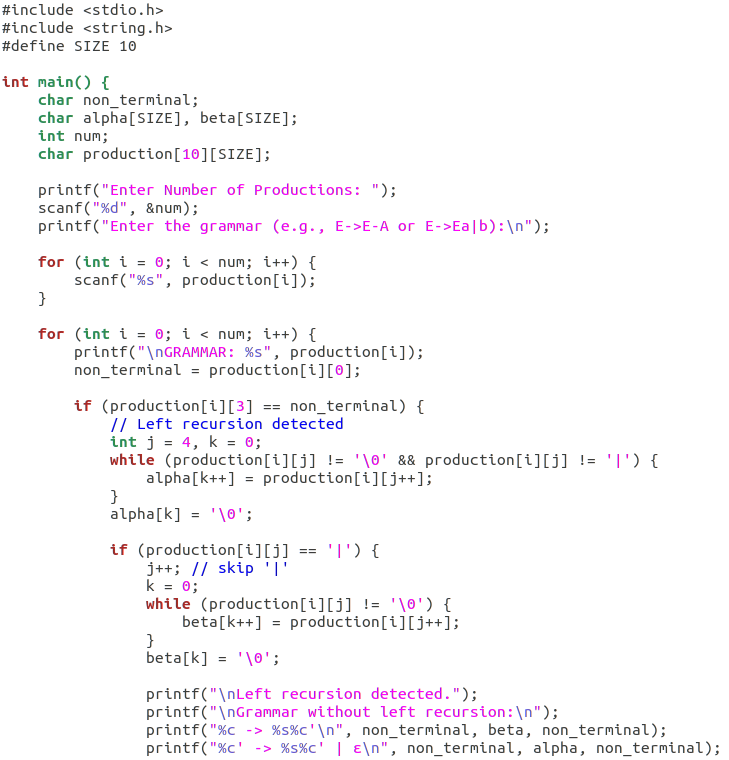
variables.

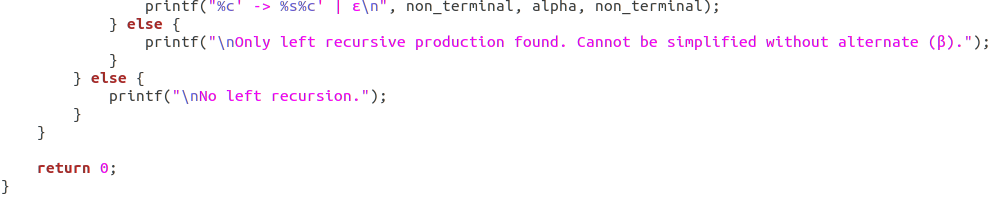
2. Check if the given grammar has left recursion.

3. Identify the alpha and beta elements in the production.

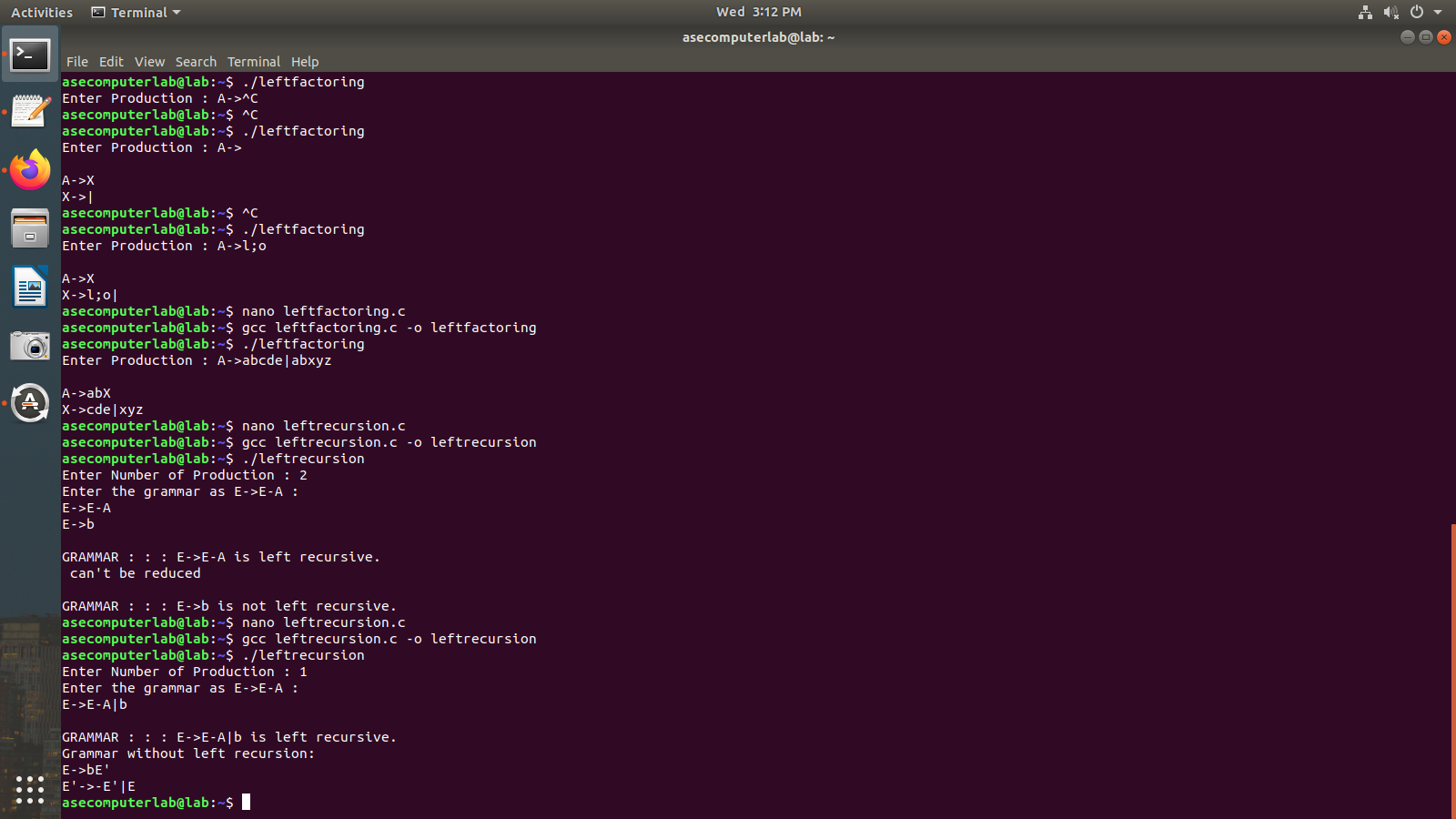
4. Print the output according to the formula to remove left recursion

**CODE:**

****

****

**OUTPUT:**

****

**RESULT:**

Thus, the program has been successfully executed.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**