**NAME : HIMASRI KOTIPALLI**

**CLASS : CSE-B**

**ROLL NO : 22122**

**COURSE : COMPILER DESIGN**

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

***BASIC LEX PROGRAMS***

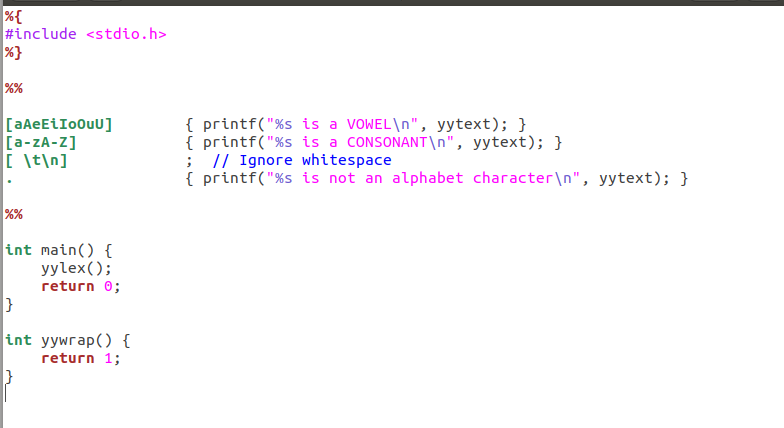
**AIM:**

Program to Identify Vowels and Consonants.

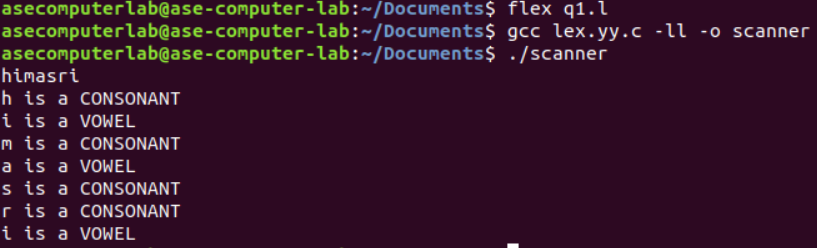
**ALGORITHM:**

* 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:**

****

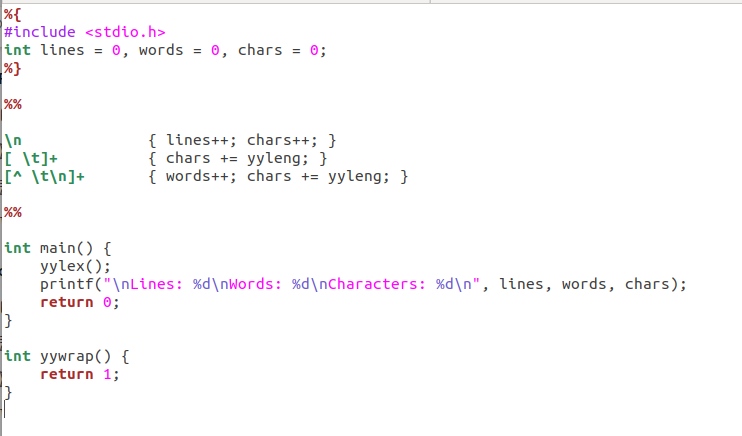
**AIM:**

Program to Count Lines, Words, and Characters.

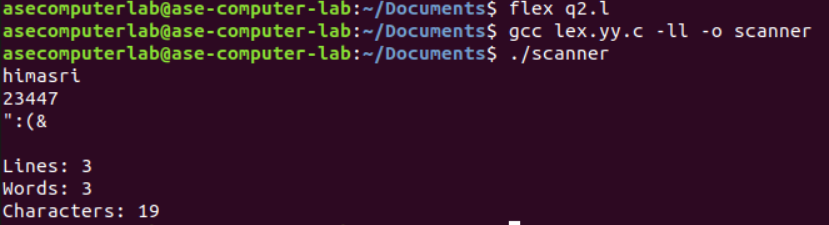
**ALGORITHM:**

* 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:**

****

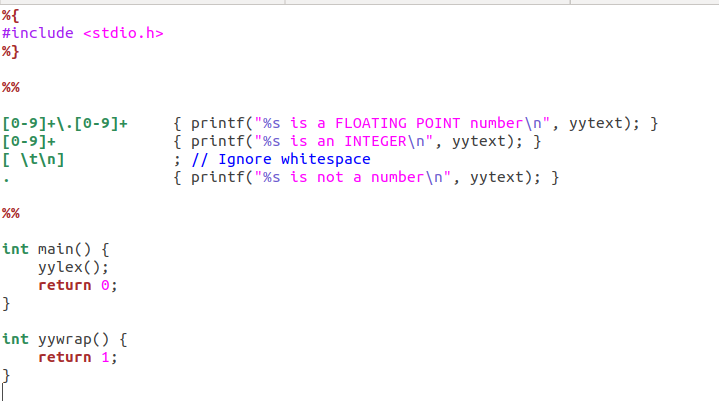
**AIM:**

Program to Recognize Integers and Floating-Point Numbers.

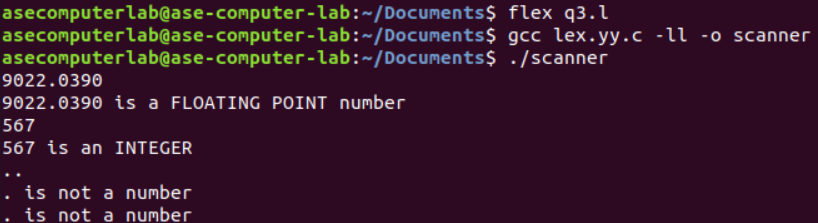
**ALGORITHM:**

* Start the Lex program and include the necessary header file.
* Define regular expressions to identify floating-point numbers, integers, whitespace, and other characters.
* Write actions to classify input as floating point, integer, or non-numeric using printf().
* Use yylex() in the main() function to start scanning the input.
* Handle the end of input using yywrap() by returning 1 to terminate scanning.

**CODE:**

****

**OUTPUT:**

****

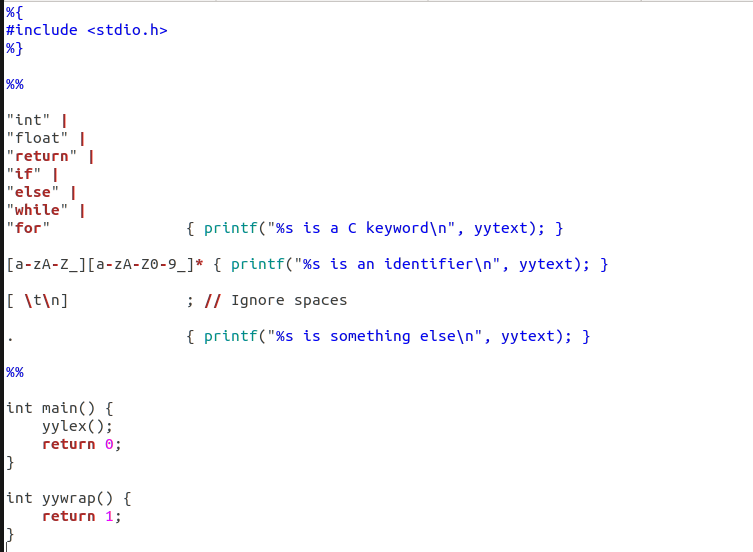
**AIM:**

Program to Recognize C Keywords.

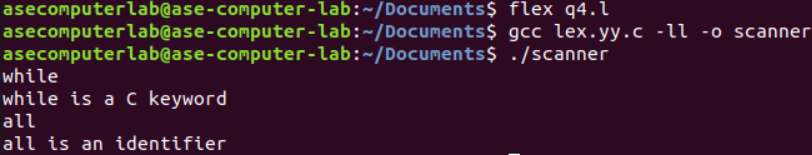
**ALGORITHM:**

* Start the Lex program and include required header files.
* Define patterns to match C keywords, identifiers, whitespace, and other characters.
* Use actions to classify matched tokens as C keywords, identifiers, or others using printf().
* Call yylex() in the main() function to begin lexical analysis.
* Terminate scanning by implementing yywrap() to return 1 at the end of input.

**CODE:**

****

**OUTPUT:**

****

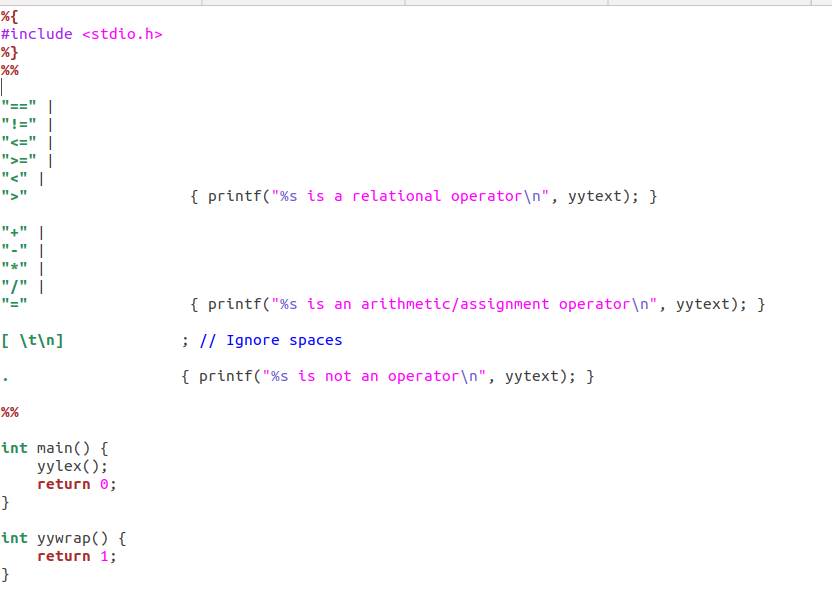
**AIM:**

Program to Recognize Operators.

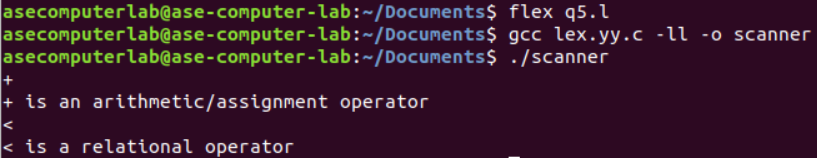
**ALGORITHM:**

* Start the Lex program and include necessary header files.
* Define regular expressions to match relational operators, arithmetic/assignment operators, whitespace, and other characters.
* Write actions to classify the matched input as relational, arithmetic/assignment, or not an operator using printf().
* Invoke yylex() in the main() function to perform lexical analysis on the input.
* Use yywrap() to signal the end of input by returning 1.

**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.

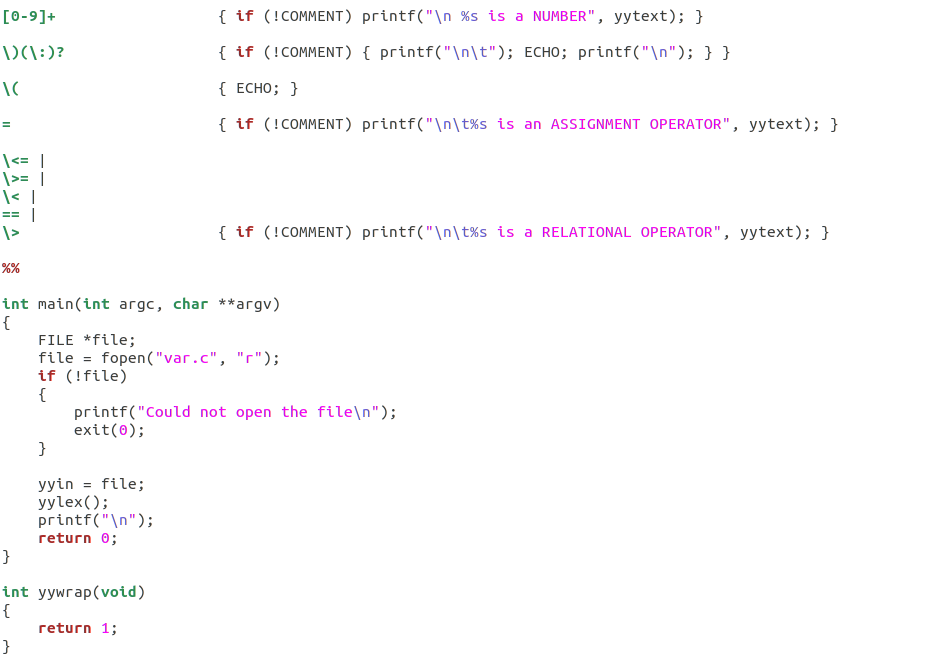
7. The Lexp.l program is run through the LEX compiler to produce an equivalent code in C language named Lex.yy.c .

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.

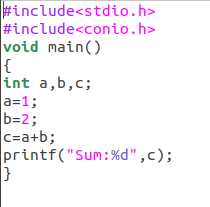
9. Finally, lex.yy.c program is run through the C Compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

**CODE:**

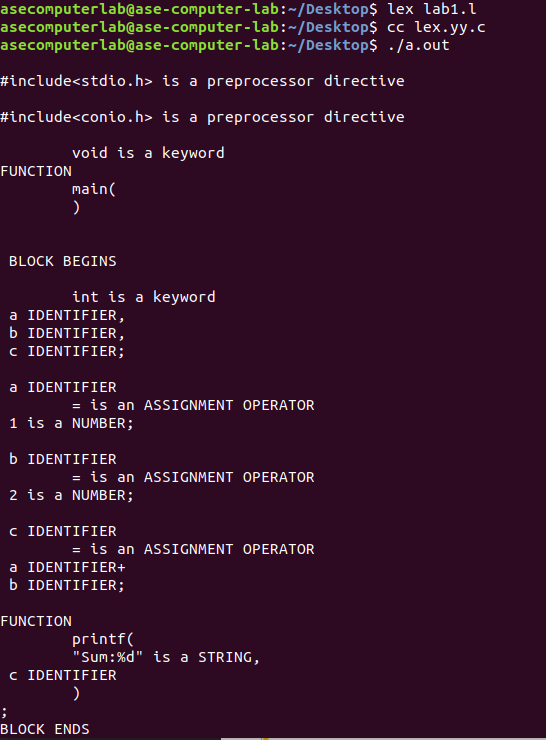
**Var.l**

****

**Var.c**

****

**OUTPUT:**



**RESULT:**

The code has been executed and output displayed successfully.

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

***LAB-2***

**AIM:**

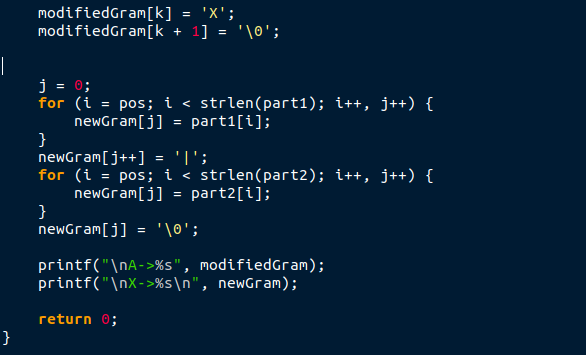
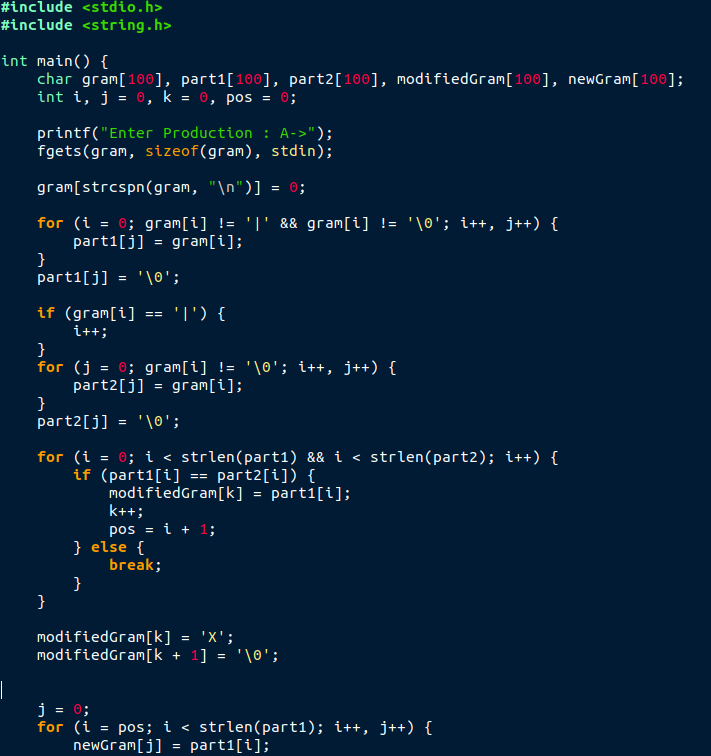
To implement eliminate left recursion and left factoring from the given grammar using C program.

**ALGORITHM:**

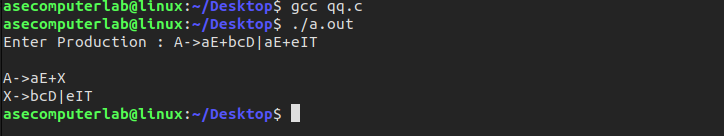
**Left Factoring:**

* 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:**

To implement LL(1) parsing using C program.

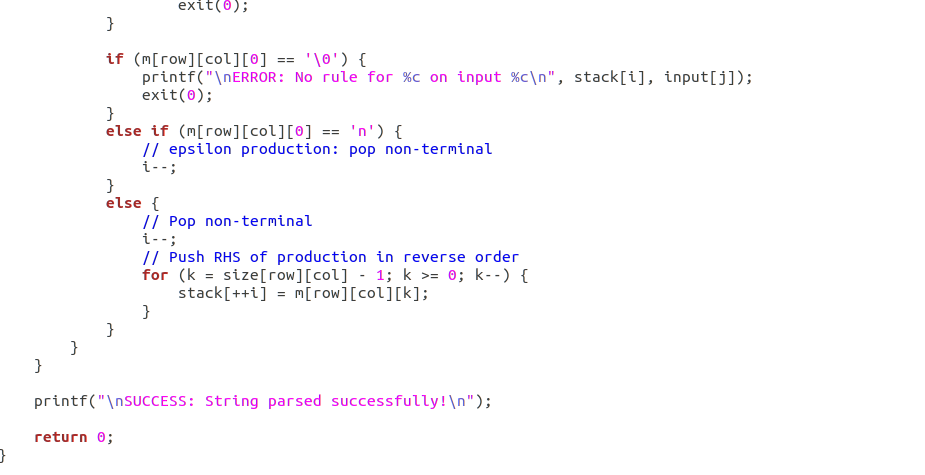
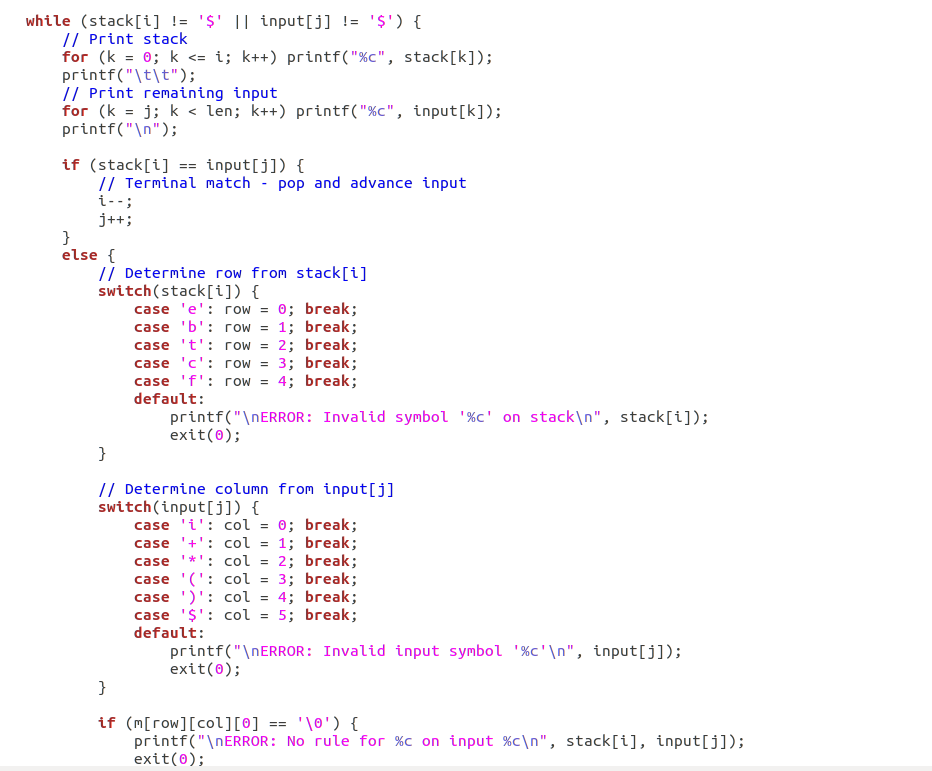
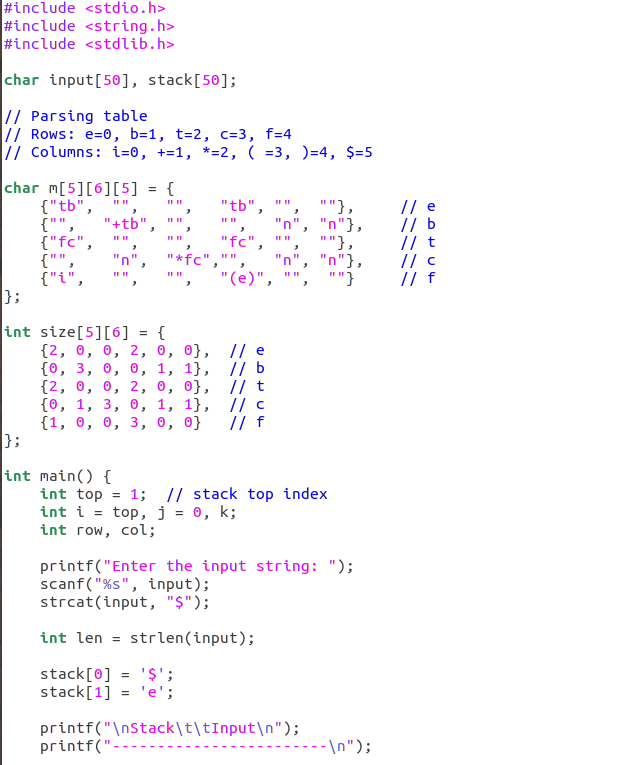
**ALGORITHM:**

1) Read the input string.

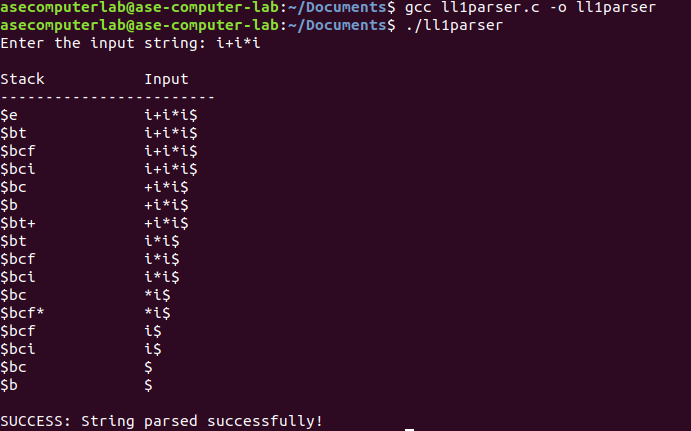
2) Using predictive parsing table parse the given input using stack.

3) If stack [i] matches with token input string pop the token else shift it repeat the process until it reaches to $.

**CODE:**



**OUTPUT:**



**RESULT:**

Thus, the program to implement LL(1) has been successfully executed.

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