

**Bandaru Mokshitha**

**CH.EN.U4CSE22171**

**INTERNET OF THINGS- 19CSE401**

**Compiler Design 19CSE401**

**Lab Report**

|  |  |  |
| --- | --- | --- |
| **Experiment No.** | **Date** | **Programs** |
| 1 | 20-Jul-2025 | Program to Identify  Vowels and Consonants |
| 2 | 25-Jul-2025 | Program to Count  Lines, Words, and  Characters |
| 3 | 30-Jul-2025 | Program to  Recognize Integers and  Floating-Point Numbers |
| 4 | 04-Aug-2025 | Program to Recognize C Keywords |
| 5 | 09-Aug-2025 | Program to Recognize  Operators |
| 6 | 14-Aug-2025 | To implement Lexical  Analyzer Using Lex Tool |
| 7 | 19-Aug-2025 | Program to eliminate left recursion and factoring  from the given grammar |
| 8 | 24-Aug-2025 | Program to eliminate left recursion and factoring  from the given grammar |
| 9 | 29-Aug-2025 | To write a program in YACC for parser generation |
| 10 | 03-Sep-2025 | To implement Symbol Table |
| 11 | 08-Sep-2025 | To implement intermediate code  generation |
| 12 | 13-Sep-2025 | To implementation of  Code Optimization  Techniques |
| 13 | 18-Sep-2025 | To write a program that implements the target  code generation |

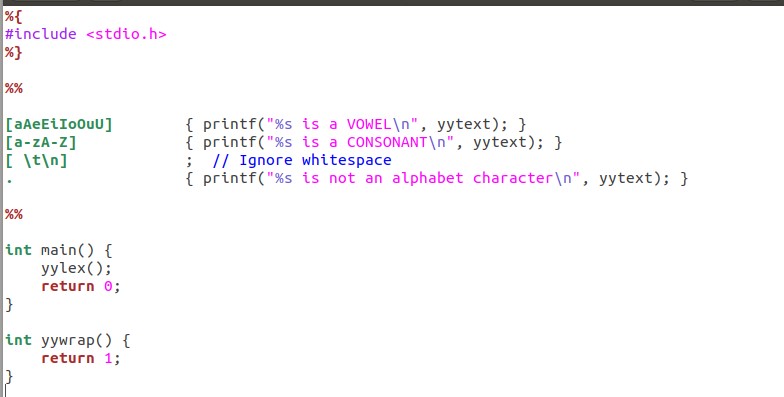
# Basic Programs

1. **Aim:** Program to Identify Vowels and Consonants

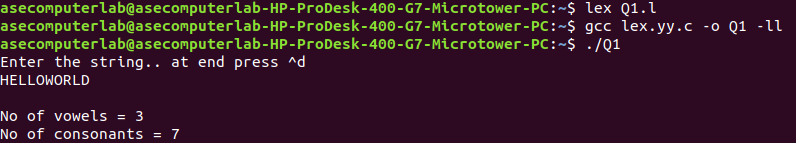
**Algorithm:**

* + Open the gedit text editor from Accessories under Applications menu.
  + Specify the header file <stdio.h> between %{ and %}.
  + Define the character patterns for vowels [aAeEiIoOuU], alphabets [a-zA-Z], whitespaces [ \t\n], and other characters ..
  + Use translation rules to print whether the character is a vowel, consonant, or not an alphabet character.
  + Call yylex() inside the main() function to begin lexical analysis.
  + Save the program as vowelconsonant.l using the LEX language.
  + Run the program using the LEX compiler to generate lex.yy.c.
  + The generated lex.yy.c contains tables and routines to match input characters.
  + Compile lex.yy.c using a C compiler to create an executable file.
  + Run the executable to check each character in the input and classify it.

**Code:**



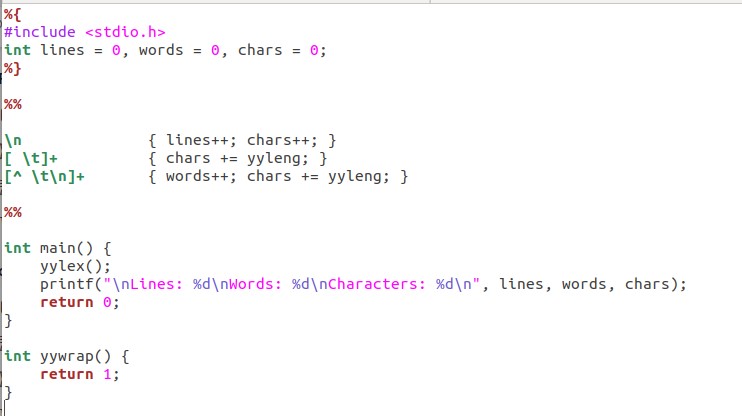
**Output:**

****

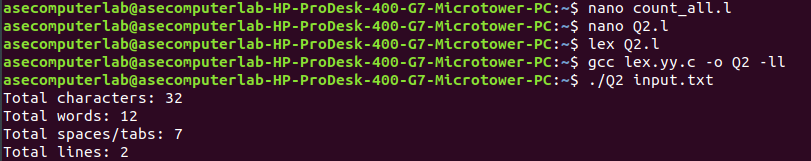
**Result:** The program has been executed successfully.

1. **Aim:** Program to Count Lines, Words, and Characters **Algorithm:** 
   * Open the gedit text editor from Accessories under Applications menu.
   * Include the header file <stdio.h> between %{ and %}.
   * Declare and initialize line, word, and character counters.
   * Define regular expressions for newline, whitespace, and words.
   * Use translation rules to update the respective counters.
   * Call yylex() inside the main() function.
   * Print the final count of lines, words, and characters.
   * Save the program as counter.l.
   * Run the program using the LEX compiler to generate lex.yy.c.
   * Compile lex.yy.c using a C compiler to produce the executable.
   * Run the executable to perform the counting operation on input.

**Code:**



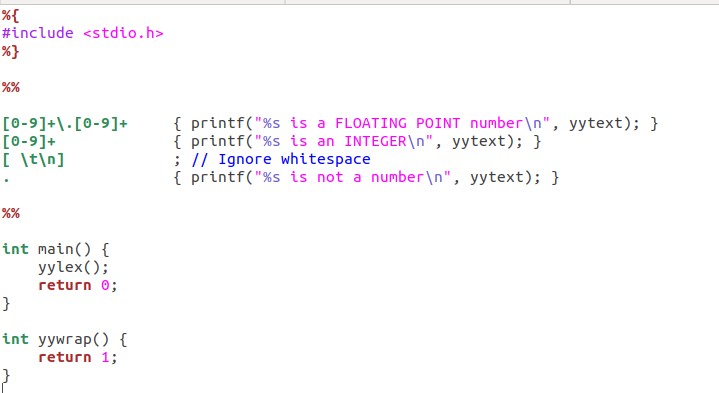
**Output:**

****

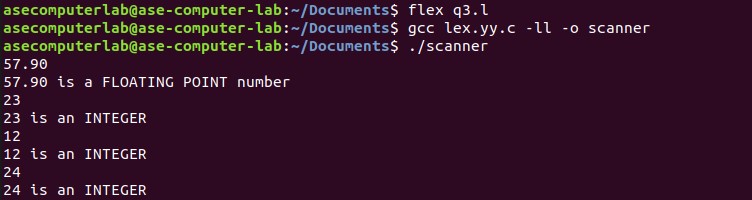
**Result:** The program has been executed successfully.

1. **Aim:** Program to Recognize Integers and Floating-Point Numbers  **Algorithm:** 
   * Open the gedit text editor from Accessories under Applications menu.
   * Include the header file <stdio.h> between %{ and %}.
   * Define patterns for floating point numbers, integers, whitespaces, and other characters.
   * Use translation rules to identify and print whether input is float, integer, or not a number.
   * Ignore whitespaces like tab, space, and newline.
   * Call yylex() inside the main() function to start lexical analysis.
   * Save the program as numcheck.l.
   * Run the program using the LEX compiler to generate lex.yy.c.
   * Compile lex.yy.c using a C compiler to get the executable.
   * Run the executable to test inputs and identify the type of number.

**Code:**



**Output:**



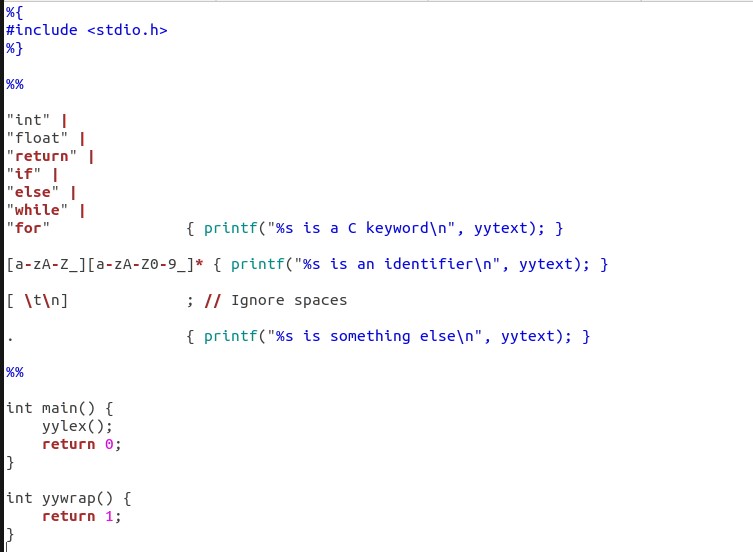
**Result:** The program has been executed successfully.

1. **Aim:** Program to Recognize C Keywords .

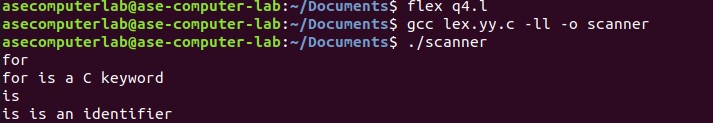
**Algorithm:**

* + Open the gedit text editor from Accessories under Applications menu.
  + Include the header file <stdio.h> between %{ and %}.
  + Define regular expressions for C keywords, identifiers, whitespaces, and other characters.
  + Use translation rules to print whether input is a C keyword, identifier, or something else.
  + Ignore spaces, tabs, and newline characters.
  + Call yylex() in the main() function to begin lexical analysis.
  + Save the program as keywordid.l.
  + Run the program through the LEX compiler to generate lex.yy.c.
  + Compile lex.yy.c using a C compiler to get the final executable.
  + Run the executable to classify each token as keyword, identifier, or other.

**Code:**



**Output:**



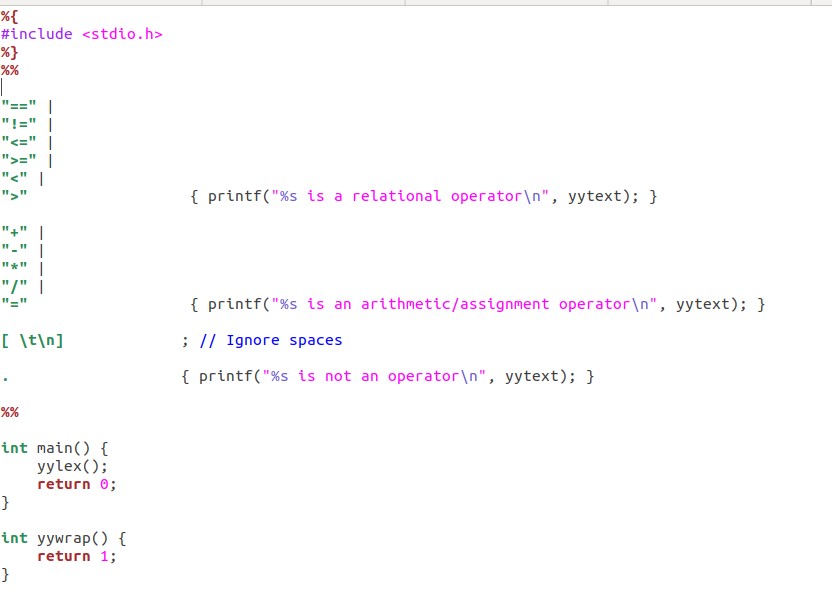
**Result:** The program has been executed successfully.

1. **Aim:** Program to Recognize Operators .

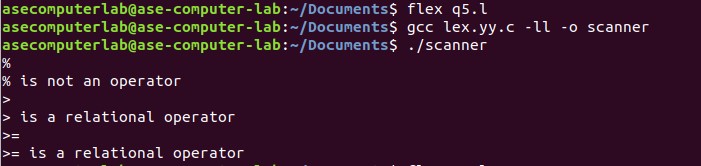
**Algorithm:**

* + Open the gedit text editor from Accessories under Applications menu.
  + Include the header file <stdio.h> between %{ and %}.
  + Define regular expressions for relational operators, arithmetic/assignment operators, whitespaces, and other characters.
  + Use translation rules to check and print whether input is a relational operator, arithmetic/assignment operator, or not an operator.
  + Ignore whitespaces like tab and newline characters.
  + Call yylex() inside the main() function to begin lexical analysis.
  + Save the program as operatorcheck.l.
  + Run the program through the LEX compiler to generate lex.yy.c.
  + Compile lex.yy.c using a C compiler to get the executable.
  + Run the executable to test and classify the input operators.

**Code:**



**Output:**



**Result:** The program has been executed successfully.

# EXPERIMENT NO – 1

**Aim:** To implement Lexical Analyzer Using Lex Tool

**Algorithm:**

* Open gedit text editor from Accessories in Applications.
* Specify the header files to be included inside the declaration part (i.e.

between %{ and %}).

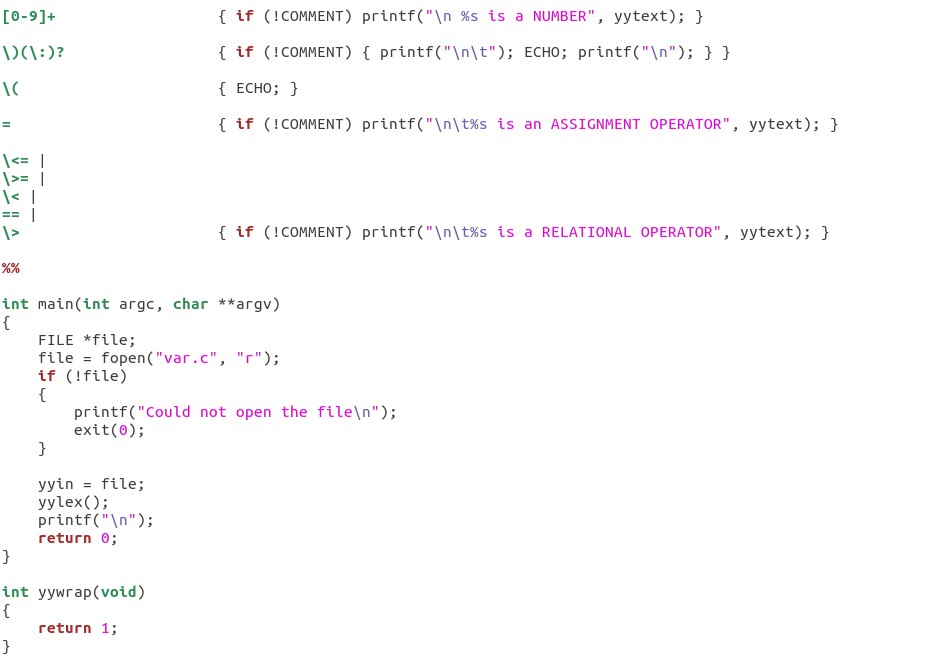
* Define the digits 0-9 and identifiers a-z and A-Z.
* Using translation rules, define the regular expressions for digit, keywords, identifiers, operators, header files etc. If matched with the input, store and display using yytext.
* Inside procedure main (), use yyin() to point to the current file being passed by the lexer. • The specification of the lexical analyzer is prepared by creating a program lab1.l in the LEX language.
* The lab1.l program is run through the LEX compiler to produce equivalent C code named lex.yy.c.
* The program lex.yy.c consists of a table constructed from the regular expressions of lab1.l, along with standard routines that use the table to recognize lexemes.
* Finally, the lex.yy.c program is run through a C compiler to produce an object program

a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

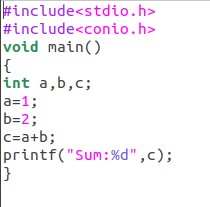
**Code:**

**Lab1.l:**

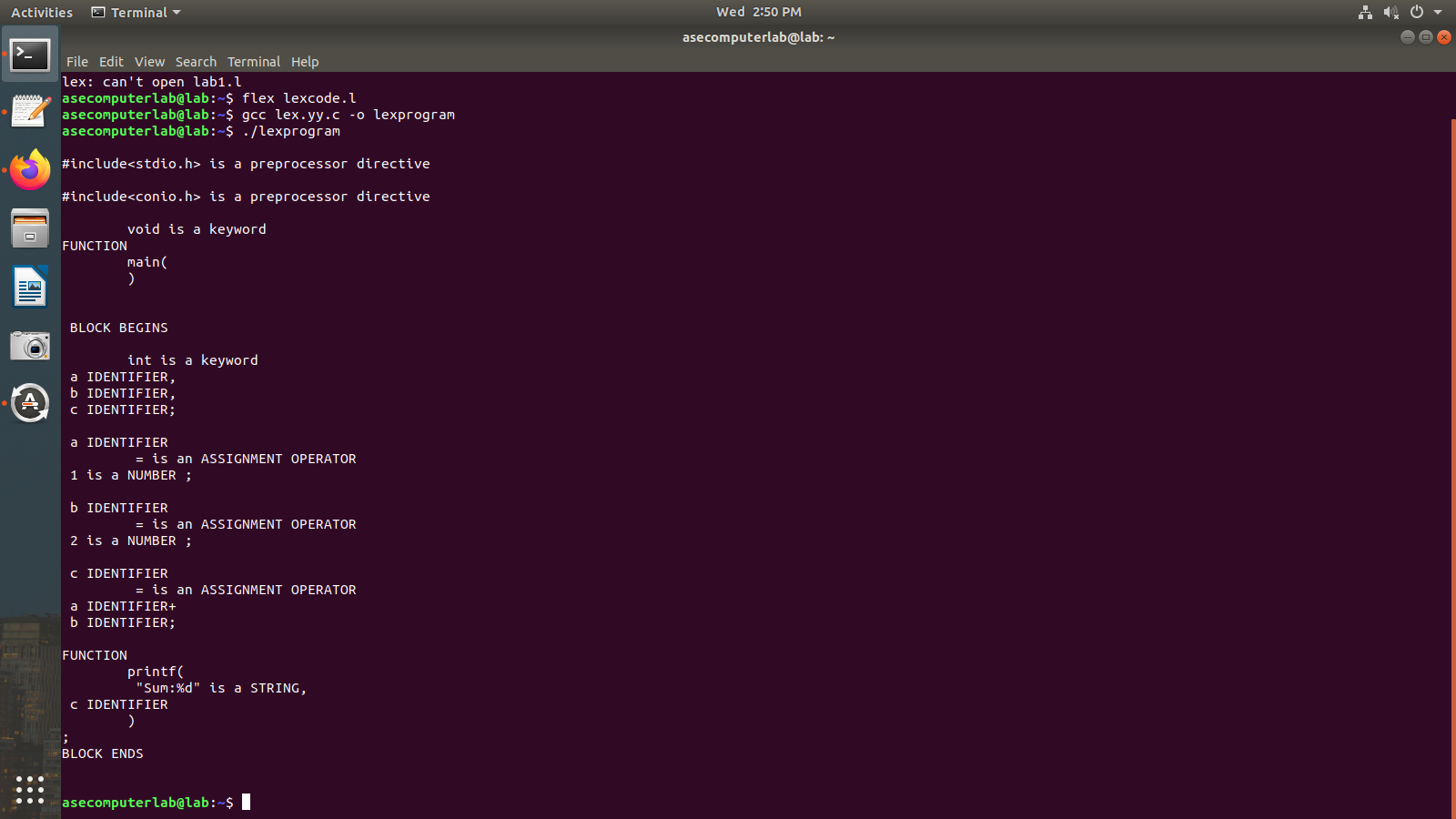




**Var.c:**



**Output:**

****

**Result:** The program has been executed successfully.

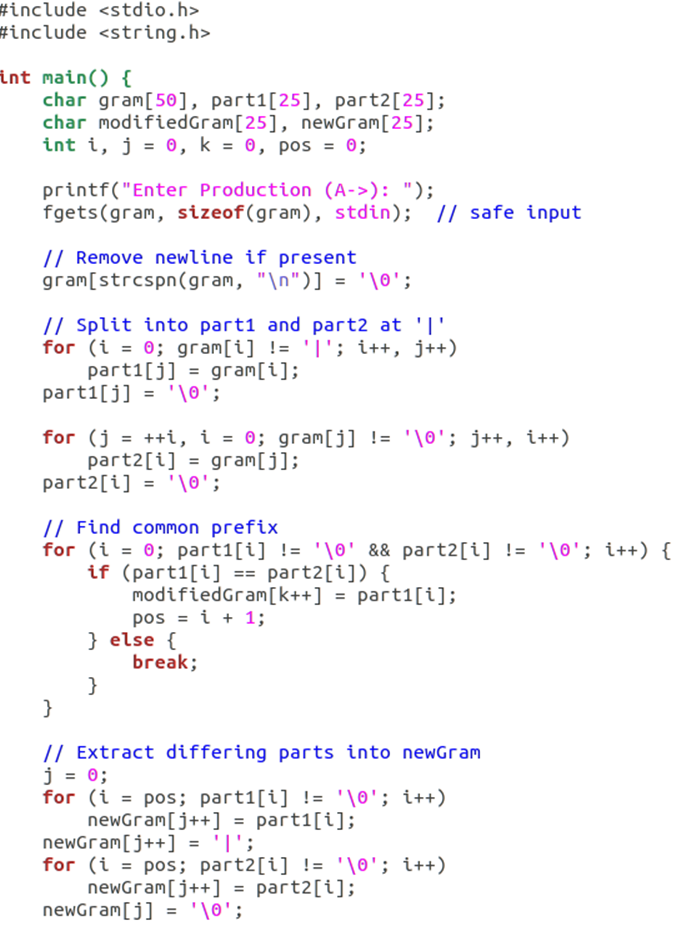
# EXPERIMENT NO – 2

**Aim:** Program to eliminate left recursion and factoring from the given grammar

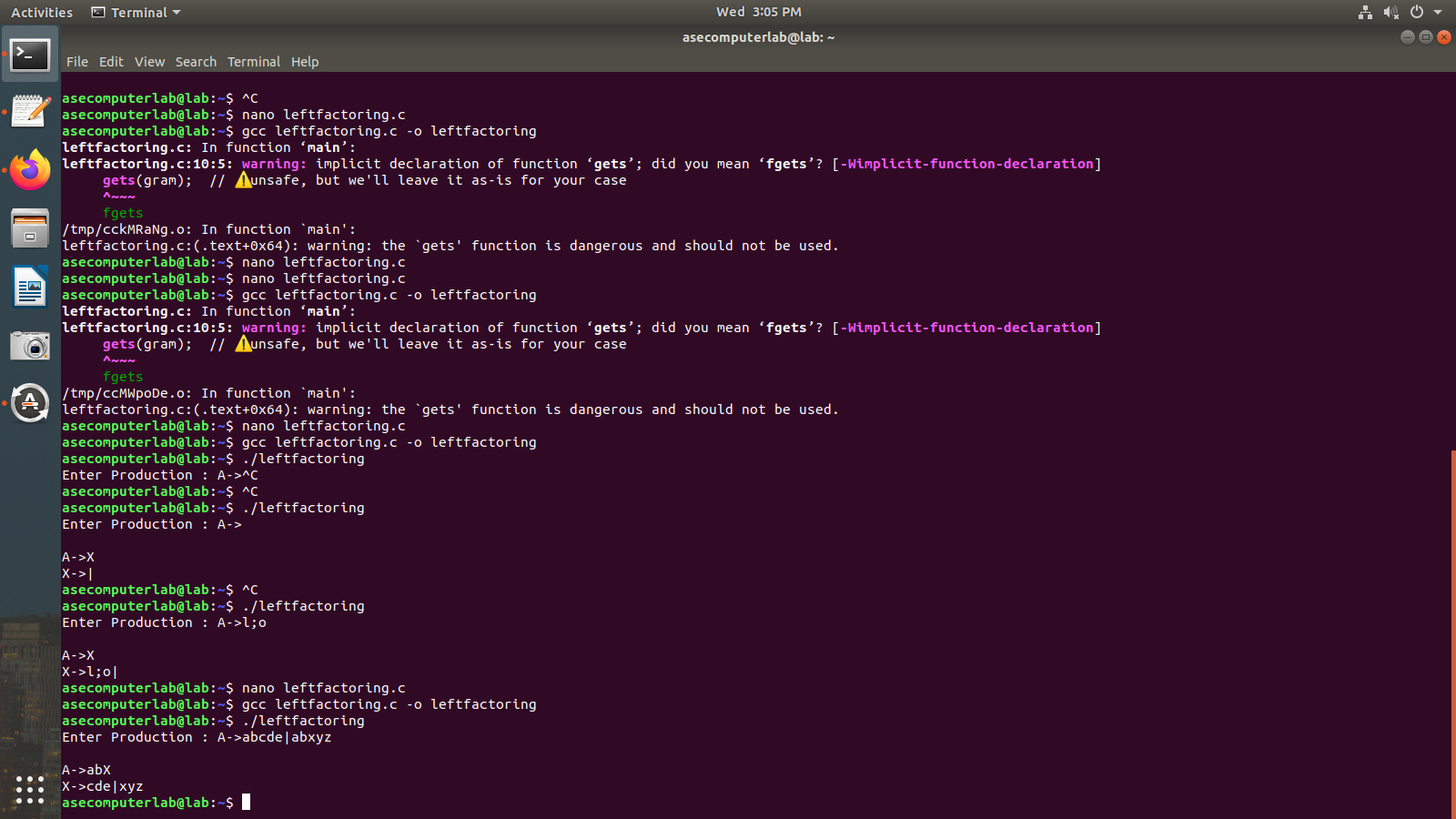
**Algorithm:**

* Open any text editor and start writing a C program.
* Include the necessary header files: stdio.h and string.h.
* Declare required character arrays for grammar parts and variables for loop counters and positions.
* Prompt the user to enter a production in the form A->alpha|beta.
* Use fgets() to read the entire input line, removing the trailing newline.
* Extract the portion before the | into part1 and the portion after into part2.
* Find the longest common prefix between part1 and part2 and store it in modifiedGram.
* After the common part, append 'X' to modifiedGram to denote the new non-terminal.
* Create newGram to store the restructured productions from the remaining suffixes of part1 and part2.
* Display the final left-factored productions using printf().

**Code:**

****

**Output:**

****

**Result:** The program has been executed successfully.

EXPERIMENT NO – 2

**AIM:** Left recursion

**Algorithm:**

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

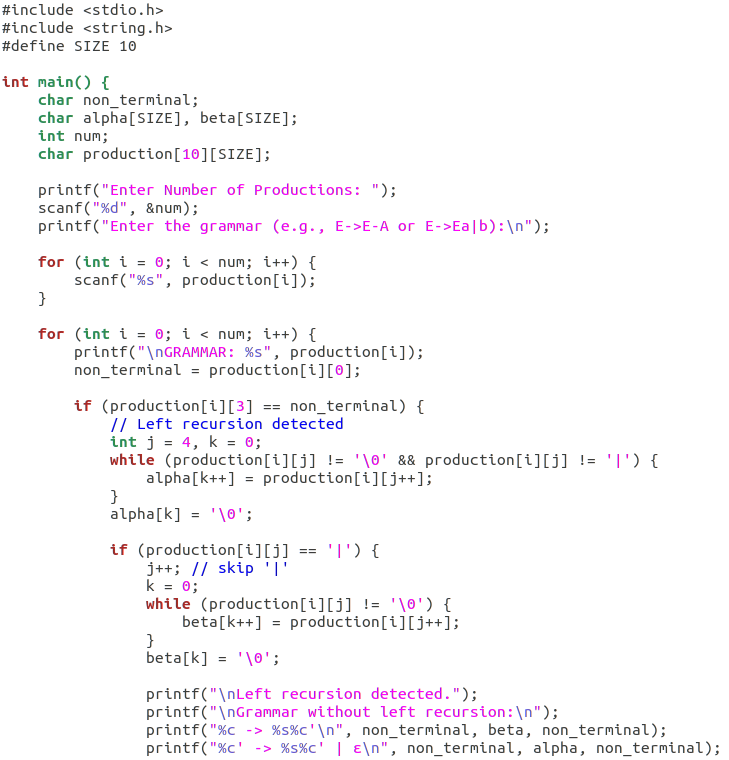
variables.

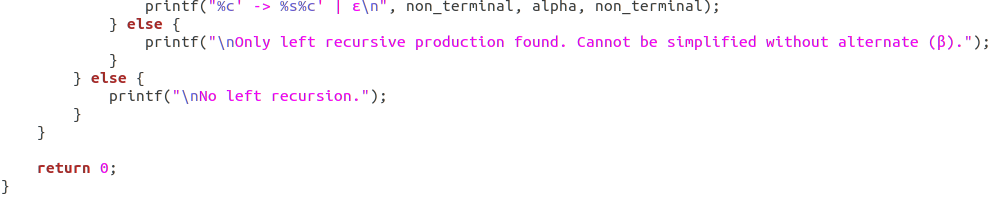
2. Check if the given grammar has left recursion.

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

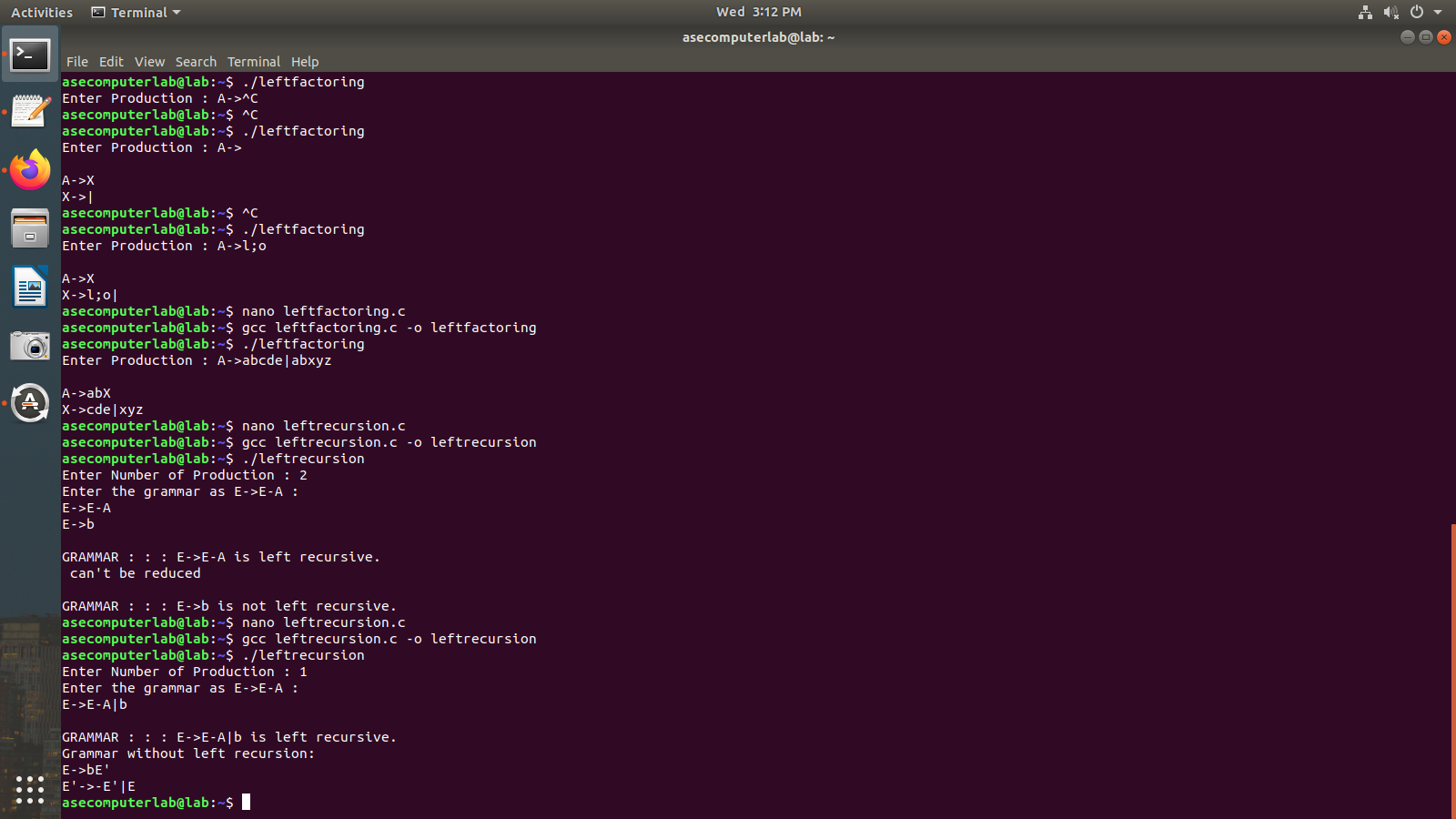
4. Print the output according to the formula to remove left recursion

**CODE:**

****

****

**OUTPUT:**

****

**RESULT:**

Thus, the program has been successfully executed.

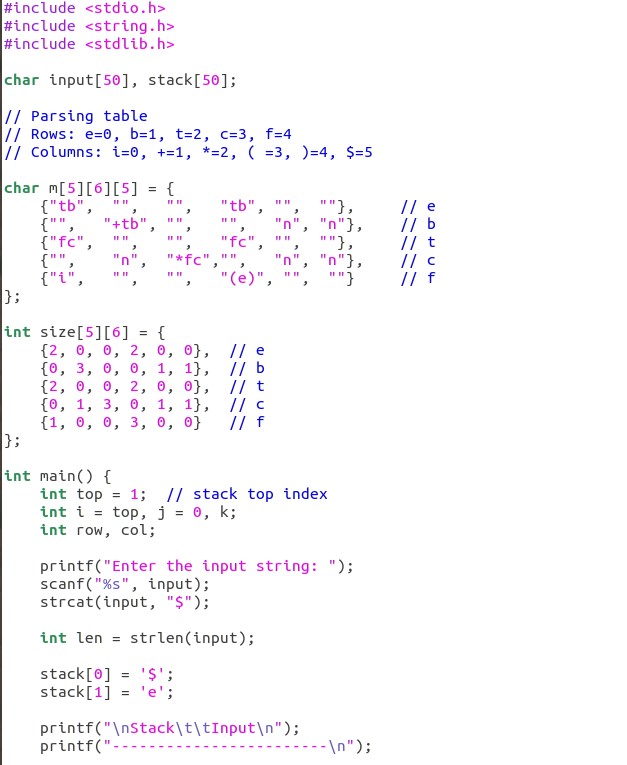
**EXPERIMENT NO – 3**

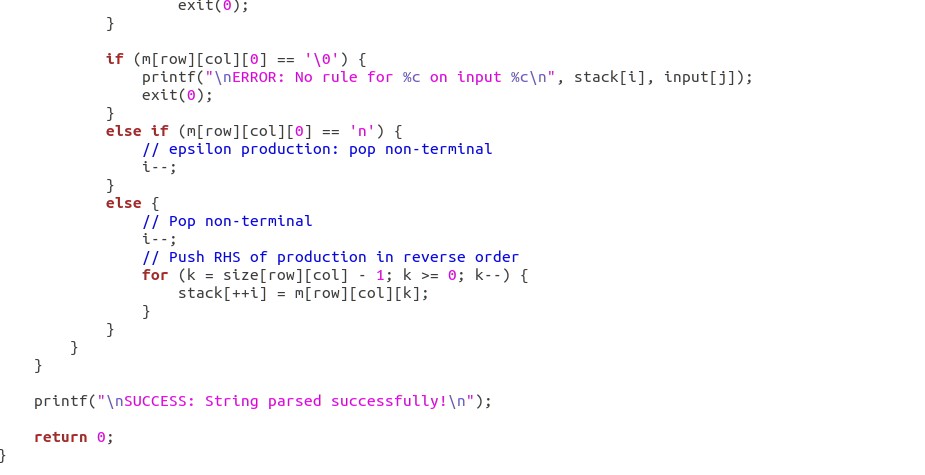
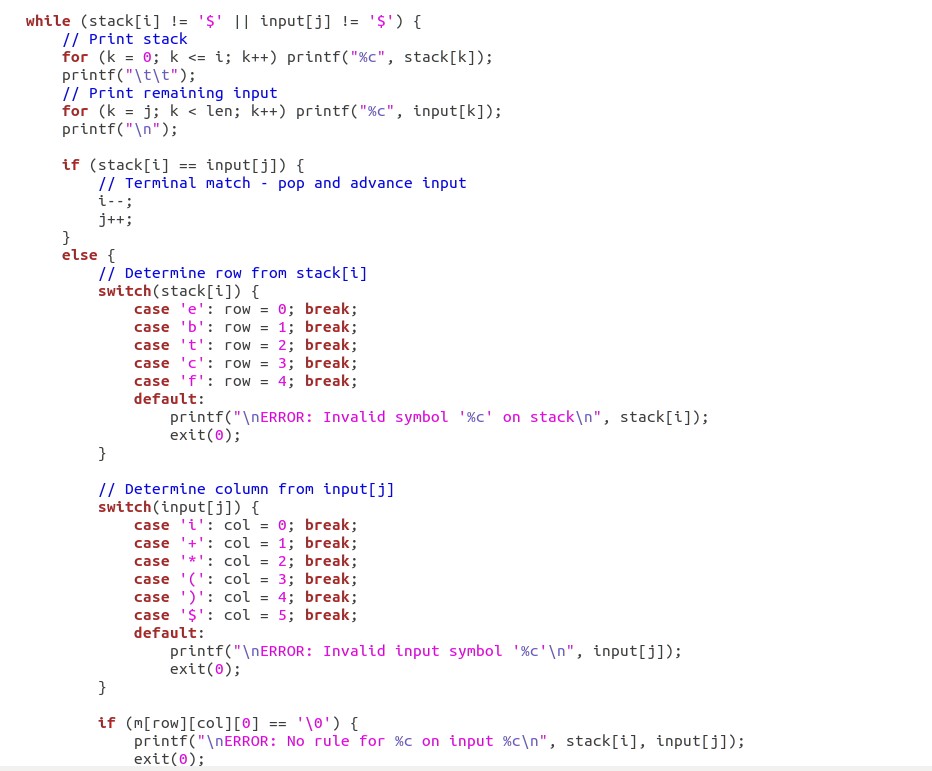
**Aim:** To implement LL(1) parsing using C program.

**Algorithm:**

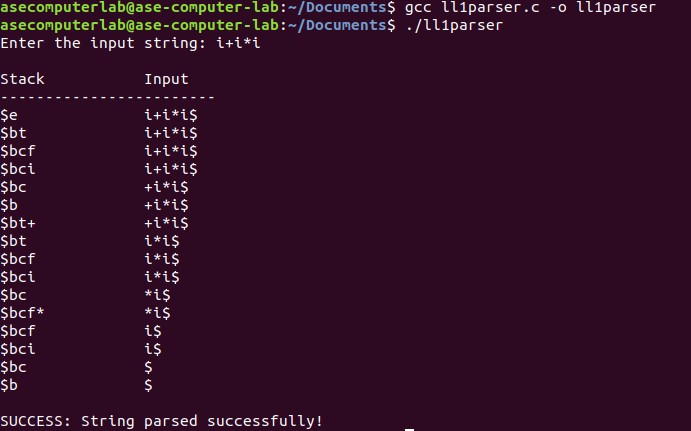
* Initialize parsing table m[][][] and size table size[][].
* Read input string from user and append '$' at the end.
* Initialize stack with '$' at the bottom and push start symbol 'e'.
* Print header for stack and input.
* Repeat until both stack top and input symbol are not '$':
* If stack top equals input symbol, pop the stack and advance input.
* Otherwise, determine row index from stack top.
* Determine column index from current input symbol.
* If no production rule exists in table, print error and exit.
* If rule is epsilon (n), pop the stack.
* If rule is a terminal like i, replace stack top with that terminal.
* Otherwise, push the right-hand side of the production rule (in reverse order) onto the stack.
* Print current contents of stack and input string.
* Continue until parsing ends.
* If successful, print “SUCCESS”.

**Code:**





**Output:**



**Result:** The program has been executed successfully.

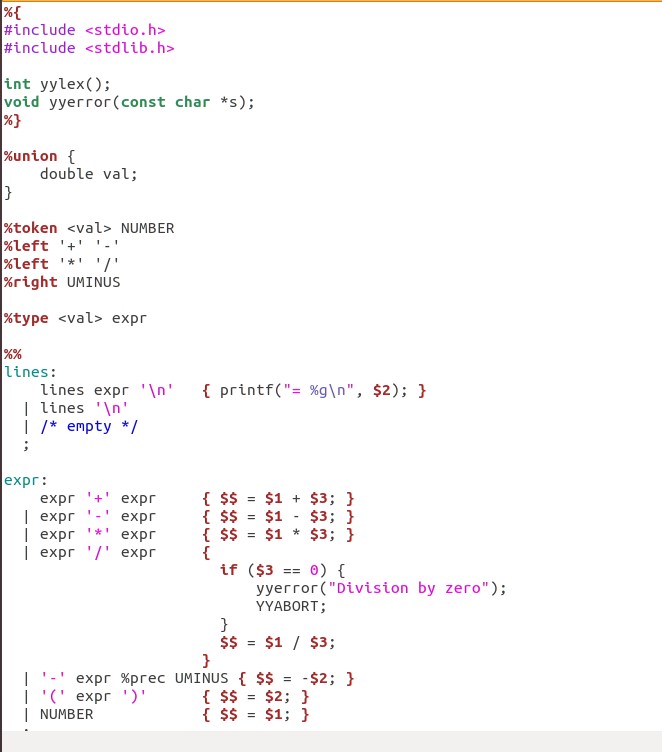
# EXPERIMENT NO – 4

**Aim:** To write a program in YACC for parser generation.

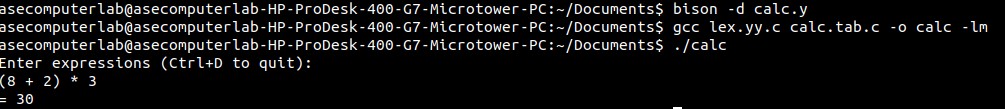
**Algorithm:**

* Start program and define grammar tokens (NUMBER, operators, parentheses) and their precedence.
* Accept input lines containing arithmetic expressions.
* Parse the expression according to grammar rules (+, -, \*, /, parentheses, unary minus, numbers).
* Perform arithmetic operations as semantic actions during parsing.
* Use yylex() to read input, skip spaces, and return tokens (numbers or operators).
* When a number is found, read it fully and assign to yylval.
* Continue parsing until the entire expression is reduced.
* Print the evaluated result of the expression and repeat for next input.

**Code:**



**Output:**



**Result:** The program has been executed successfully.

# EXPERIMENT NO – 5

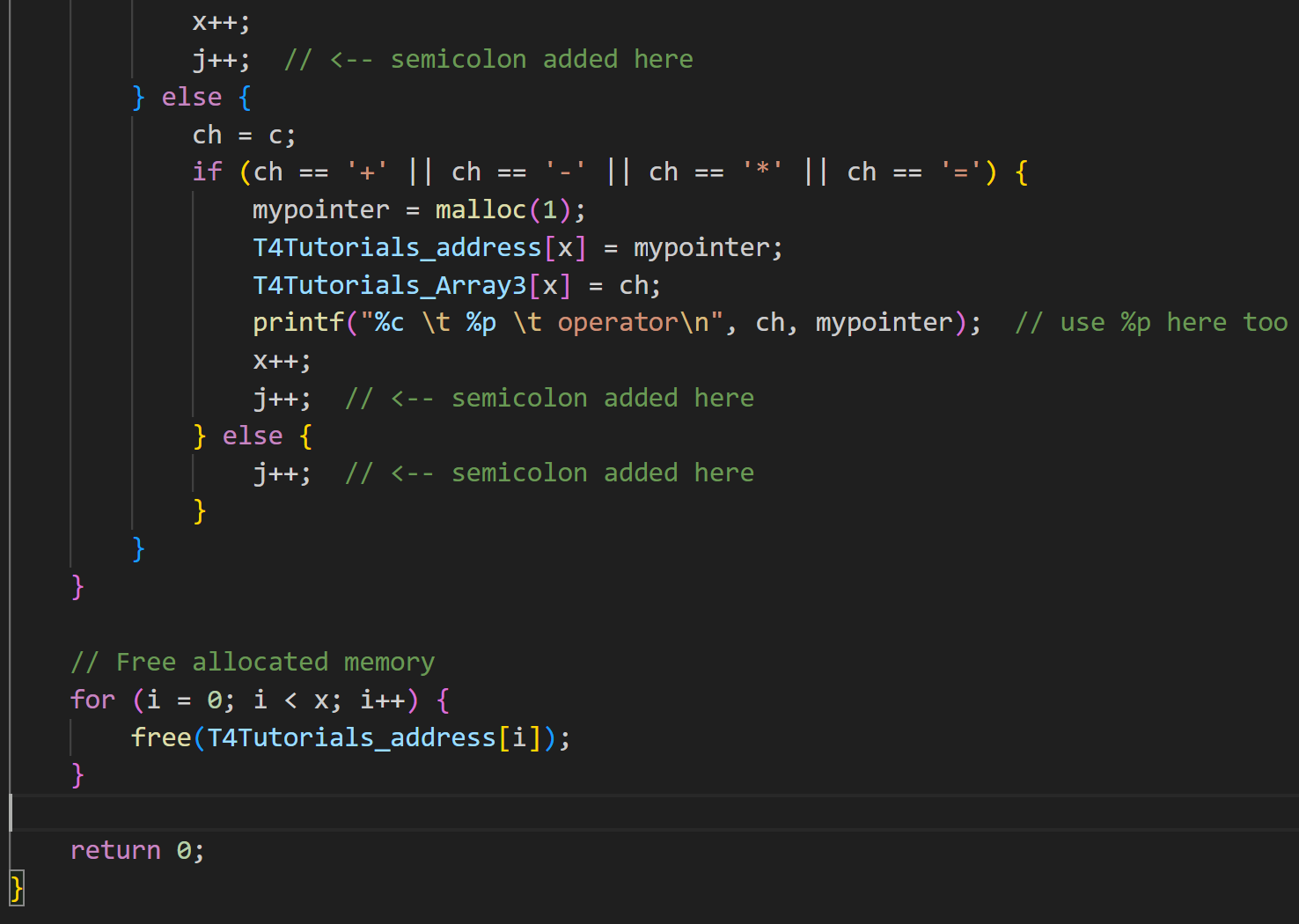
**Aim:** To implement Symbol Table.

**Algorithm:**

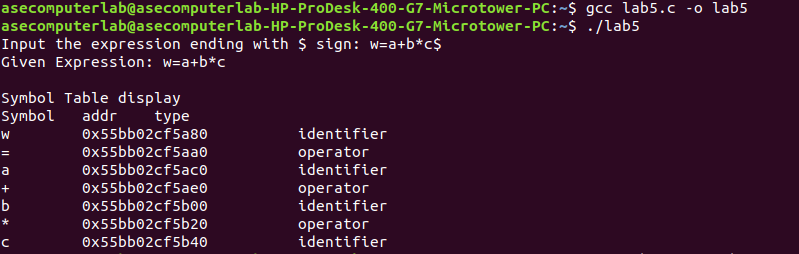
* Start the program and read an expression ending with $.
* Store the input characters into an array.
* Display the given expression.
* Traverse each character of the expression.
* If the character is an alphabet, classify it as an identifier and store with its address.
* If the character is an operator (+, -, \*, =), classify it as an operator and store with its address.
* Display the complete symbol table and end the program.

**Code:**





**Output:**



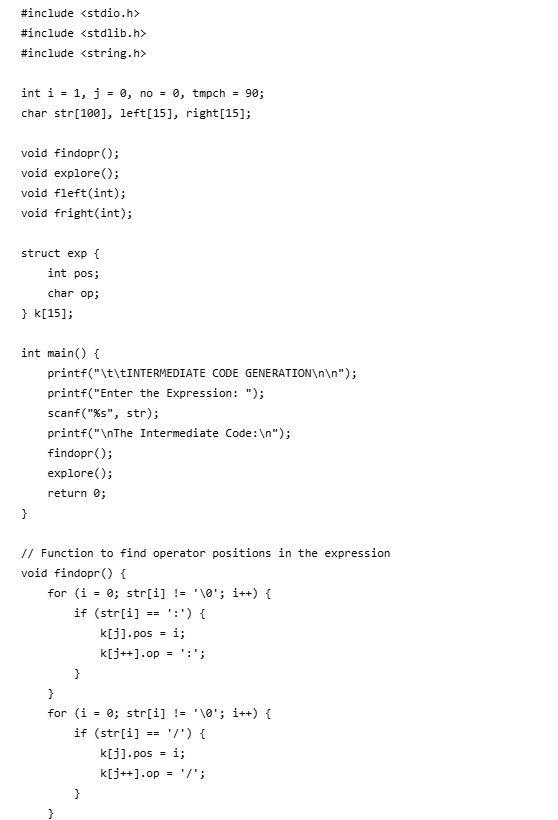
**Result:** The program has been executed successfully.

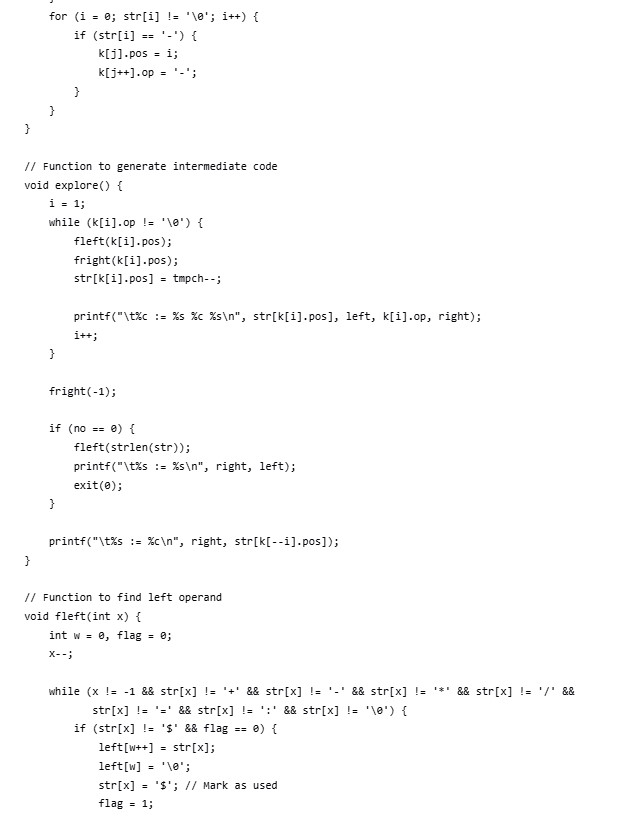
# EXPERIMENT NO – 6

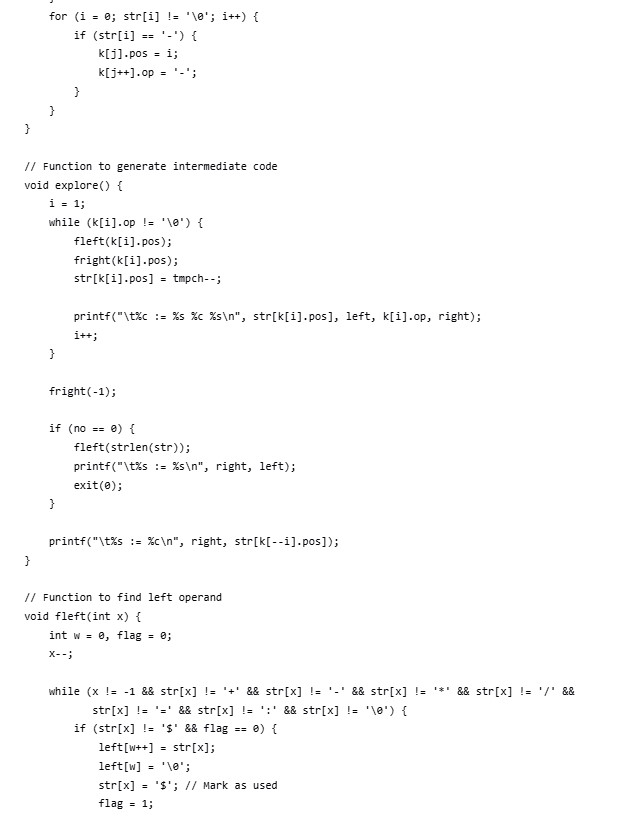
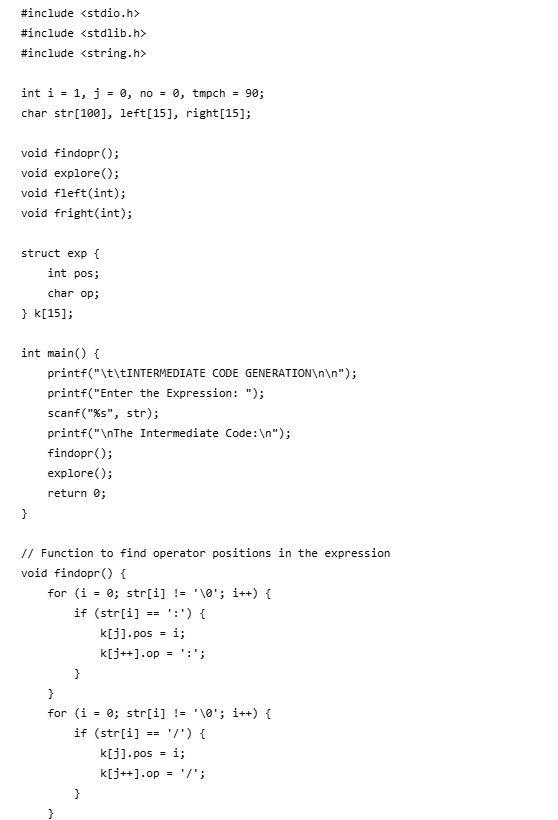
**Aim:** To implement intermediate code generation.

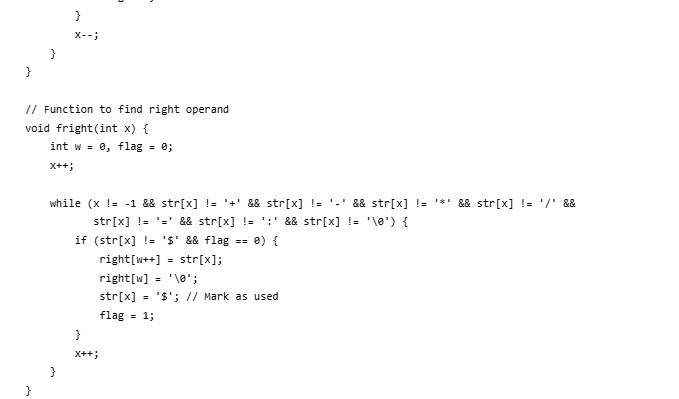
**Algorithm:**

* Start the program and read an arithmetic expression as input.
* Scan the expression and record the positions of operators (:, /, \*, +, -).
* For each operator, find its left operand and right operand.
* Generate a temporary variable for the result and replace the operator with it.
* Print the intermediate code in the form of three-address statements (T := operand1 op operand2).
* Repeat the process until the full expression is reduced.
* Print the final assignment statement and end the program.

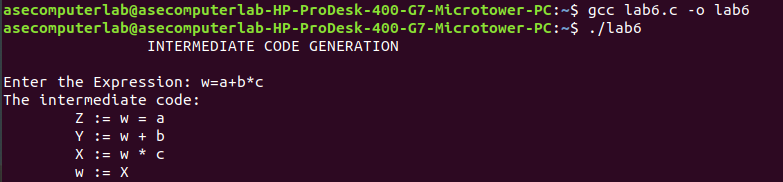
**Code:**







**Output:**

****

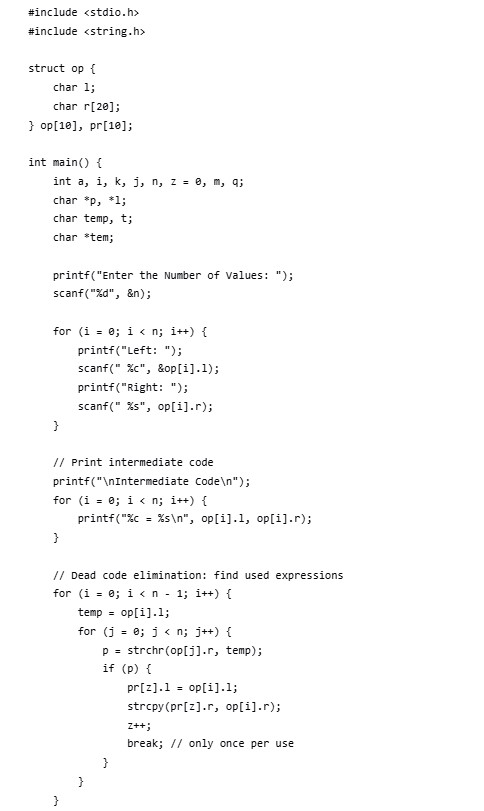
**Result:** The program has been executed successfully.

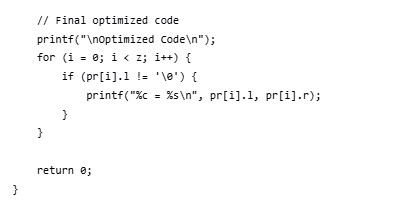
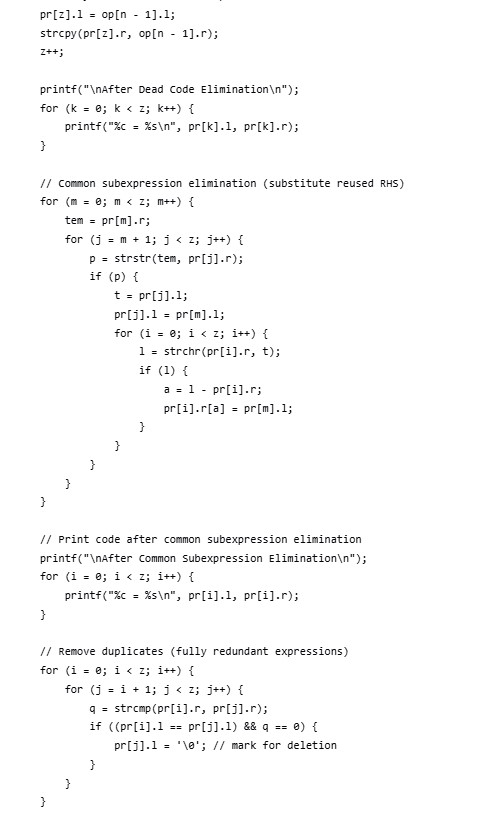
**EXPERIMENT NO – 7**

**Aim:** To implementation of Code Optimization Techniques **Algorithm:**

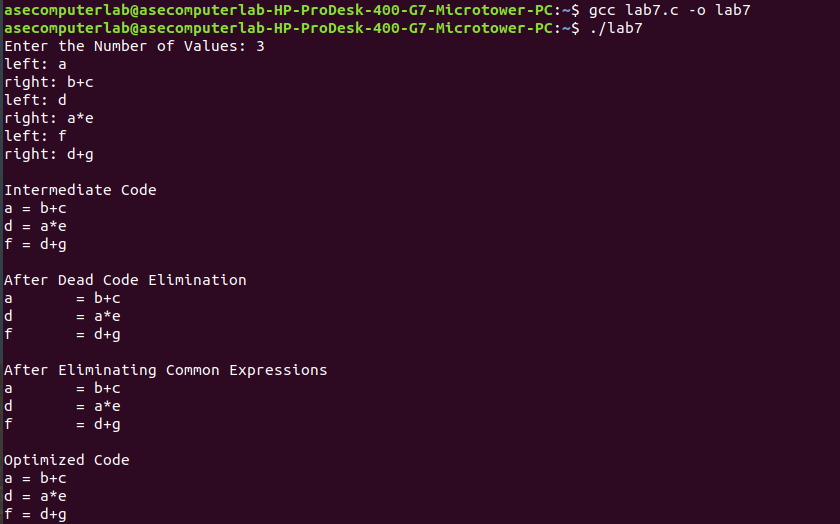
* Start the program and read the number of expressions (n).
* For each expression, input the left-hand side variable and the right-hand side expression.
* Display the original intermediate code.
* Perform dead code elimination by keeping only those statements whose results are used later.
* Perform common subexpression elimination by checking if two expressions compute the same value and replacing duplicates.
* Update references so that redundant variables are replaced with the optimized variable.
* Print the final optimized code and end the program.

**Code:**





**Output:**



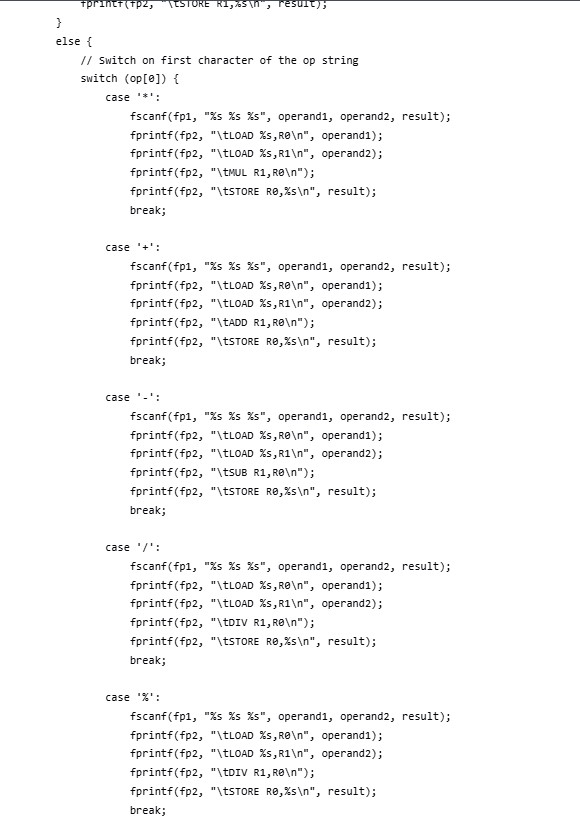
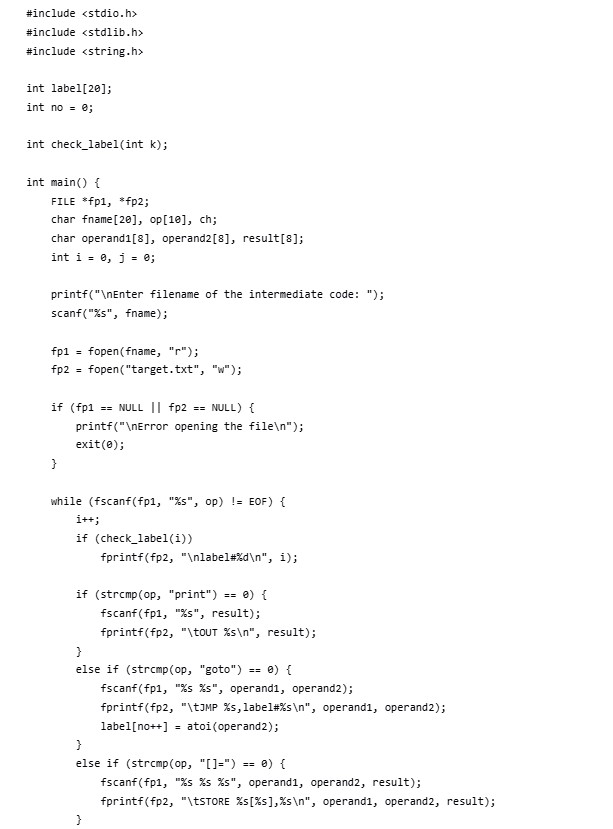
**Result:** The program has been executed successfully.

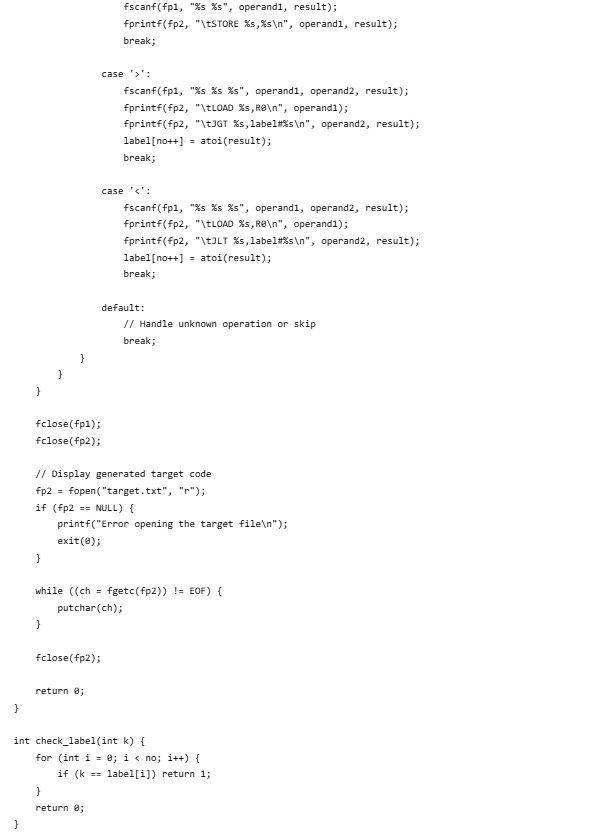
# EXPERIMENT NO – 8

**Aim:** To write a program that implements the target code generation **Algorithm:**

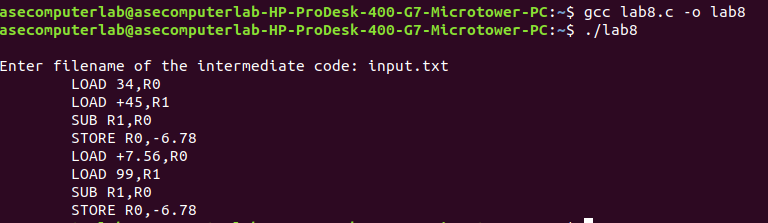
* Read the input string from the user.
* Process each input string and use a switch–case structure to identify the operator.
* Load the input variables into temporary variables (operands) and display them using the instruction LOAD.
* Based on the arithmetic operator, display the corresponding operation (ADD, SUB, MUL, DIV) using switch–case.
* Generate the three-address code representation for each operation.
* If the operator is an assignment (=), store the result in the target variable and display it using STORE.
* Repeat this process for each line of the input string.
* Display the final output, which is the transformed assembly-like machine code.

**Code:**





**Output:**



**Result:** The program has been executed successfully.