**LAB RECORD**

**COMPILER DESIGN**

**19CSE401**



**Amrita School of Computing**

**Amrita Vishwa Vidyapeetham**

**Chennai – 601 103, Tamil Nadu, India.**

**BONAFIDE CERTIFICATE**

**University Reg. No** : \_\_\_\_\_\_\_CH.EN.U4CSE22030\_\_\_\_\_\_

This is to certify that this is a bonafide record work done by

Mr. \_\_\_\_\_\_\_\_Keerthi Rohan\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ studying

B.Tech\_\_\_\_\_\_\_\_\_\_4th Year\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

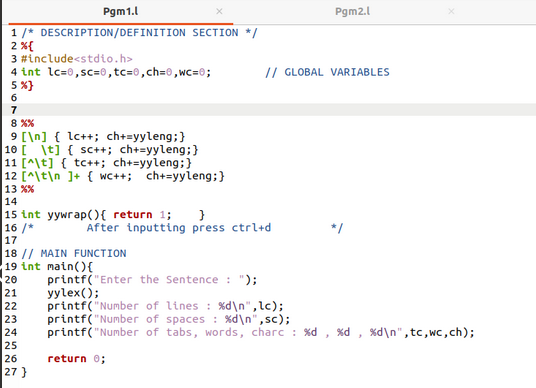
**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **Experiment Number** | **Topic Name** | **Date** | **Page no** |
| **01** | **Implementation of Lexical Analyzer Using Lex Tool** | 7-07-2025 | 4 |
| **02** | **Program to eliminate left recursion and factoring from the given grammar** | 21-07-2025 | 9 |
| **03** | **Implementation of LL(1) parsing** | 11-09-2025 | 12 |
| **04** | **Parser Generation using YACC** | 25-09-2025 | 14 |
| **05** | **Implementation of Symbol Table** | 1-09-2025 | 16 |
| **06** | **Implementation of Intermediate Code Generation** | 8-09-2025 | 18 |
| **07** | **Implementation of Code Optimization Techniques** | 15-09-2025 | 20 |
| **08** | **Implementation of Target code generation** | 22-09-2025 | 22 |

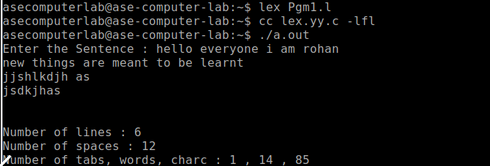
**Exercise 1**

1. Aim : Write a Lex program to count number of lines, spaces and etc.

Code :

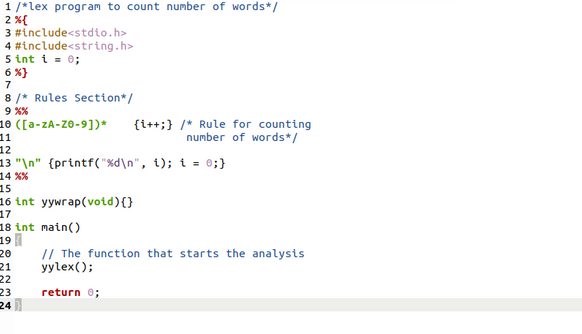


Output :

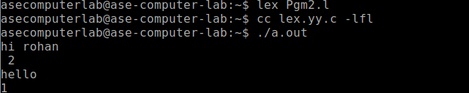


1. Aim : Write a Lex program to count number of words in given sentence.

Code :

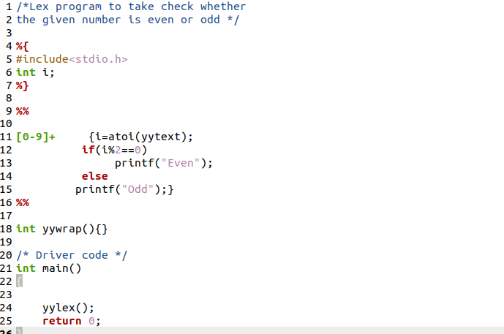


Output :

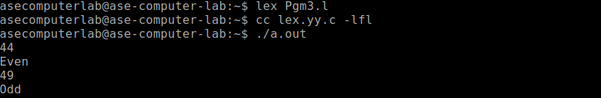


1. Aim : Write a Lex program to check whether the given number is even or odd

Code :



Output :

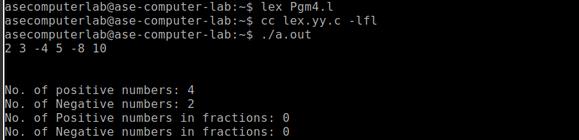


1. Aim : Write a Lex program to count the positive numbers, negative numbers and fractions.

Code :

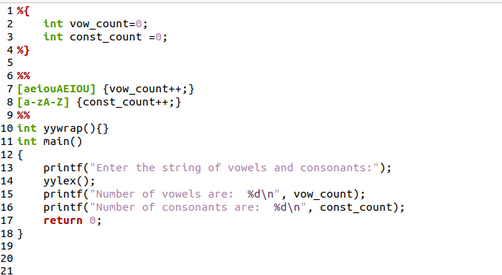


Output :

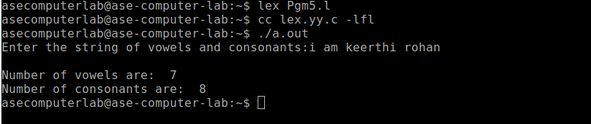


1. Aim : Write a Lex program to count the vowels and consonants in the given string

Code :



Output :



**Exercise 2**

1. 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

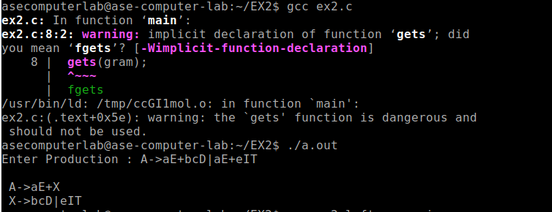
separate grammar

* Display the new and modified grammar.

Code:



Output:



Left recursion:

1. Aim: To implement left recursion using C.

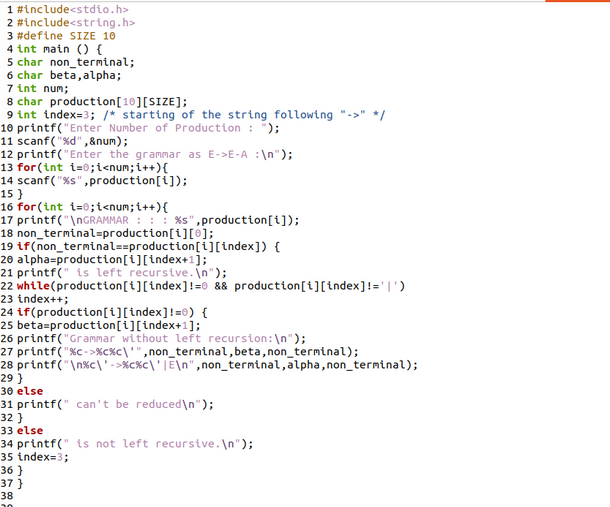
Algorithm:

* Start the processes by getting the grammar and assigning it to the appropriate

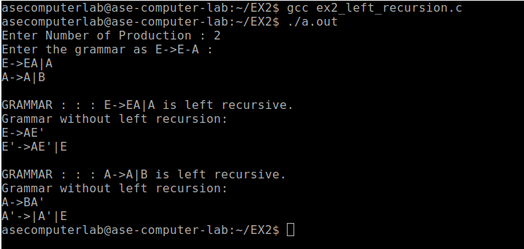
variables.

* Check if the given grammar has left recursion.
* Identify the alpha and beta elements in the production.
* Print the output according to the formula to remove left recursion

Code:



Output:



Result: The program to implement left factoring and left recursion has been successfully executed.

**Exercise 3**

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

Algorithm:

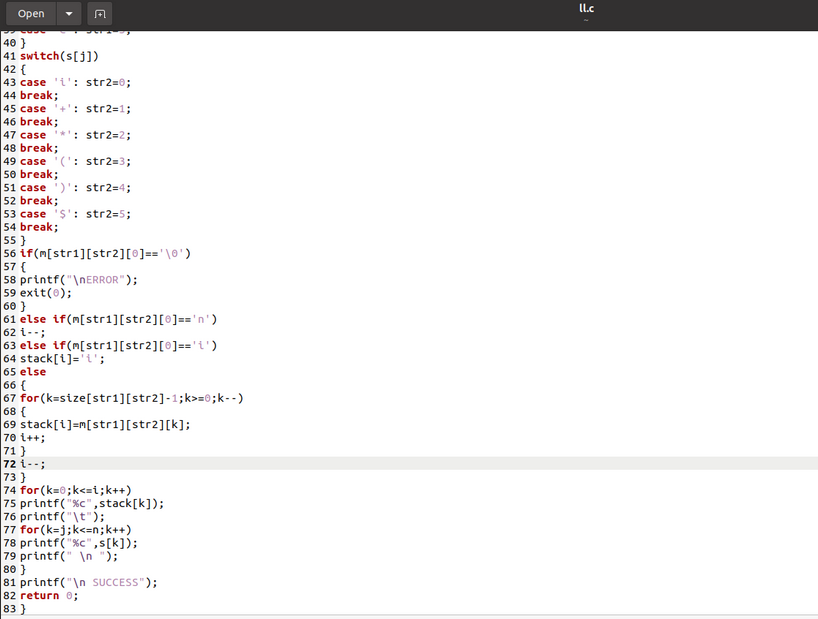
1) Read the input string.

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

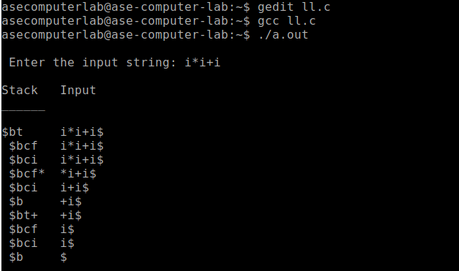
3) If stack [i] matches with token input string pop the token else shift it repeat the

process until it reaches to $.

Code :



Output :



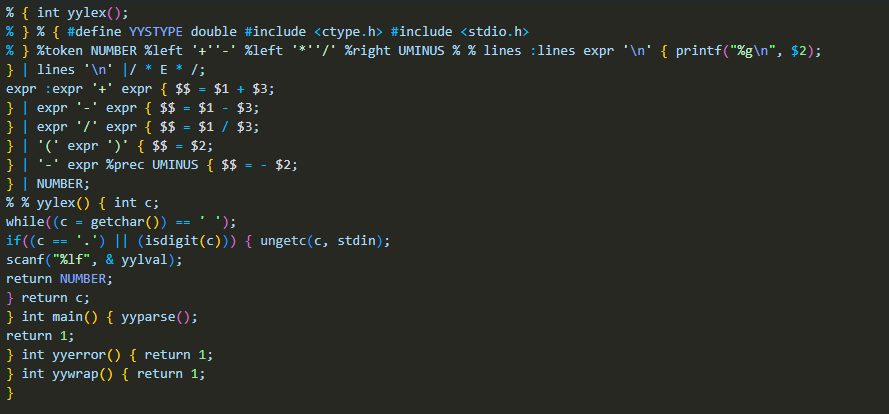
**Exercise 4**

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

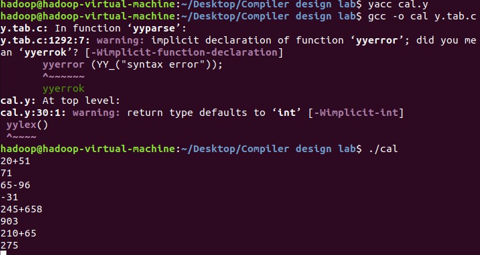
Algorithm:

1. Get the input from the user and Parse it token by token.
2. First identify the valid inputs that can be given for a program.
3. Define the precedence and the associativity of various operators like +,-,/,\* etc.
4. Write codes for saving the answer into memory and displaying the result on the screen.
5. Write codes for performing various arithmetic operations.
6. Display the possible Error message that can be associated with this calculation.
7. Display the output on the screen else display the error message on the screen

Code :



Output :



**Result:** Thus the program in YACC for parser generation has been executed successfully.

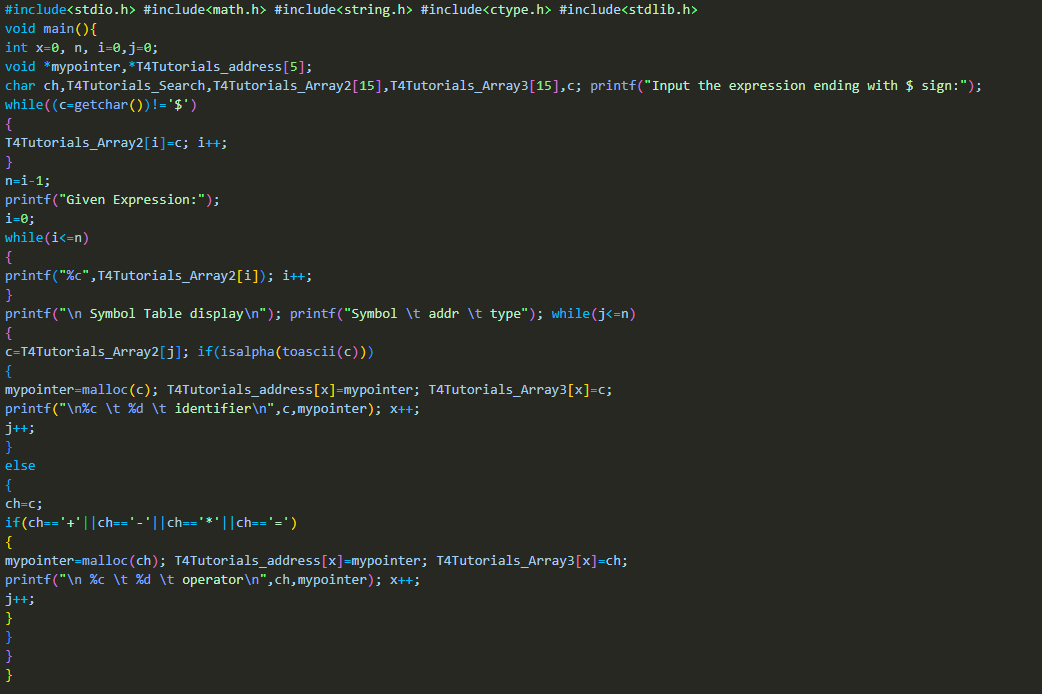
**Exercise 5**

Aim: To implement Symbol Table**.**

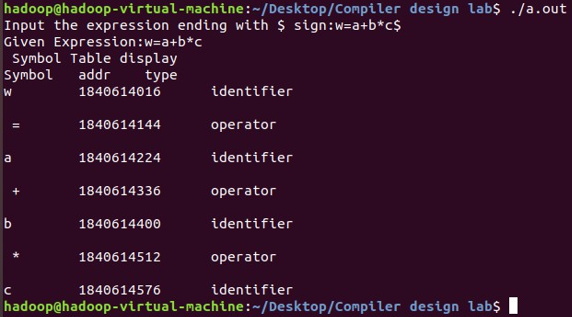
Algorithm:

1. Start the Program.
2. Get the input from the user with the terminating symbol ‘$’.
3. Allocate memory for the variable by dynamic memory allocation function.
4. If the next character of the symbol is an operator then only the memory is allocated.
5. While reading, the input symbol is inserted into symbol table along with its memory address.
6. The steps are repeated till “$” is reached.
7. To reach a variable, enter the variable to the searched and symbol table has been
8. Checked for corresponding variable, the variable along its address is displayed as result.
9. Stop the program

Code :



Output :



**Result:** Thus the program to implement symbol table has been executed successfully.

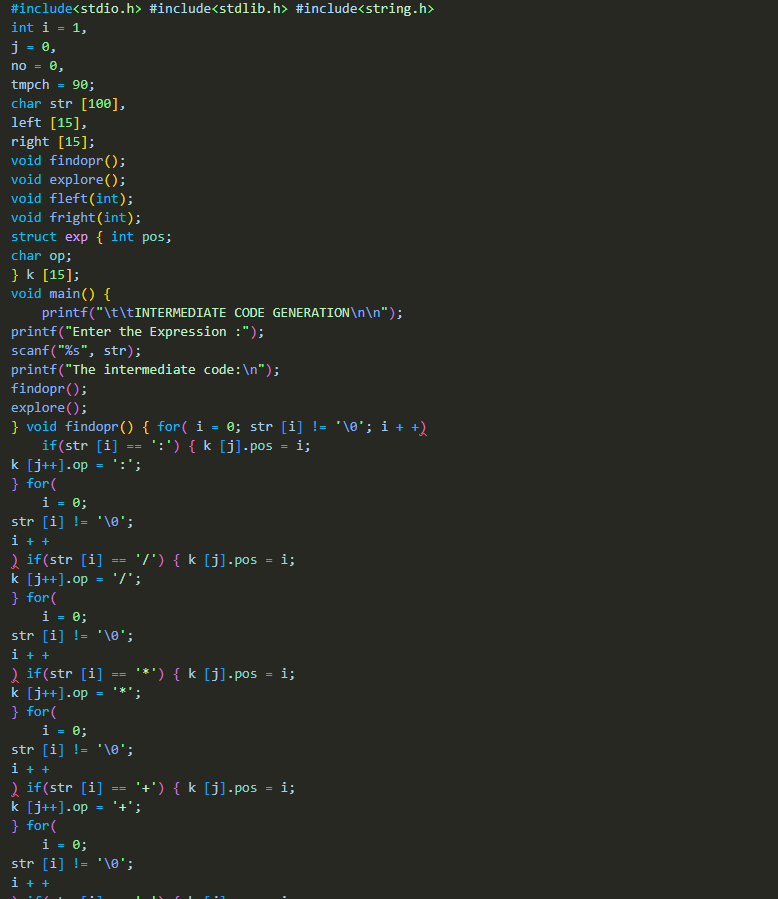
**Exercise 6**

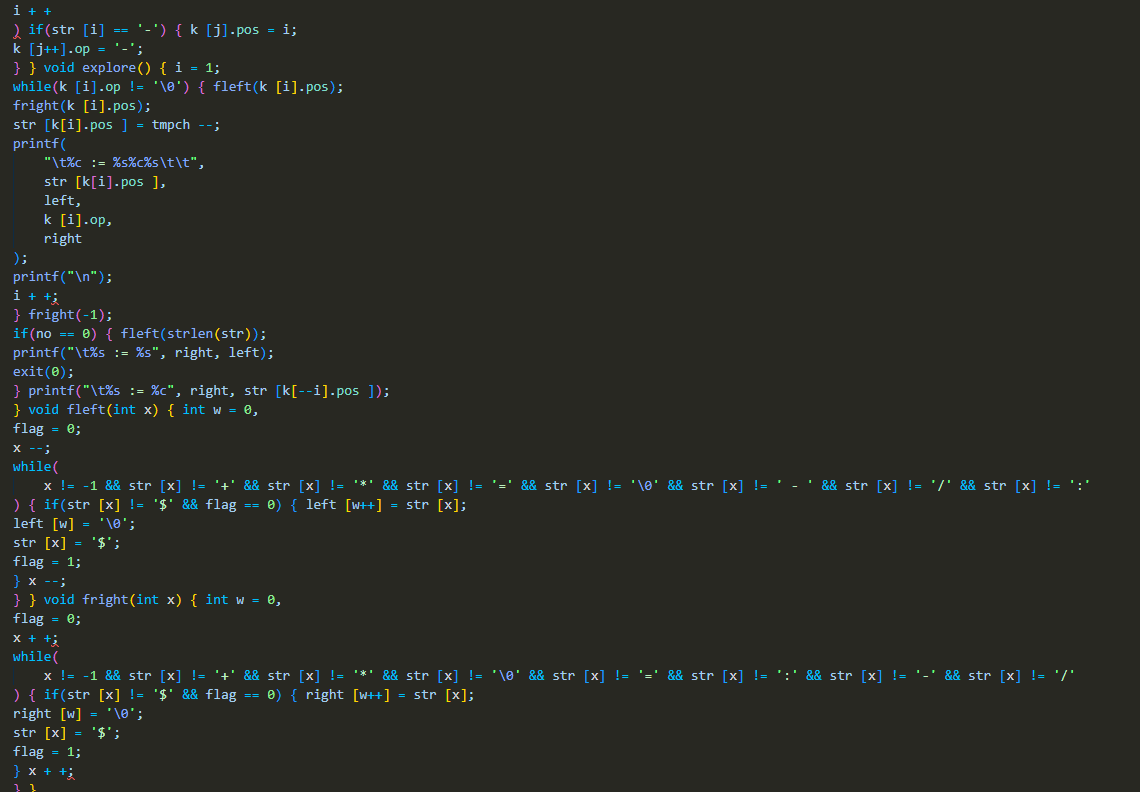
Aim: To implementation of intermediate code generation.

Algorithm:

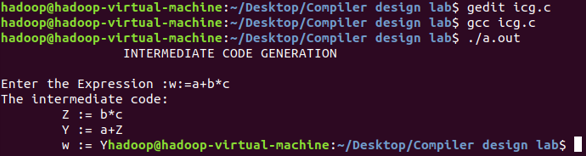
1. Take the parse tree tokens from the syntax analyser.
2. Generate intermediate code using temp variable
3. Assign the final temp variable to initial variable

Code :





Output :



**Result:** Thus, the program to implement intermediate code generation has been executed successfully.

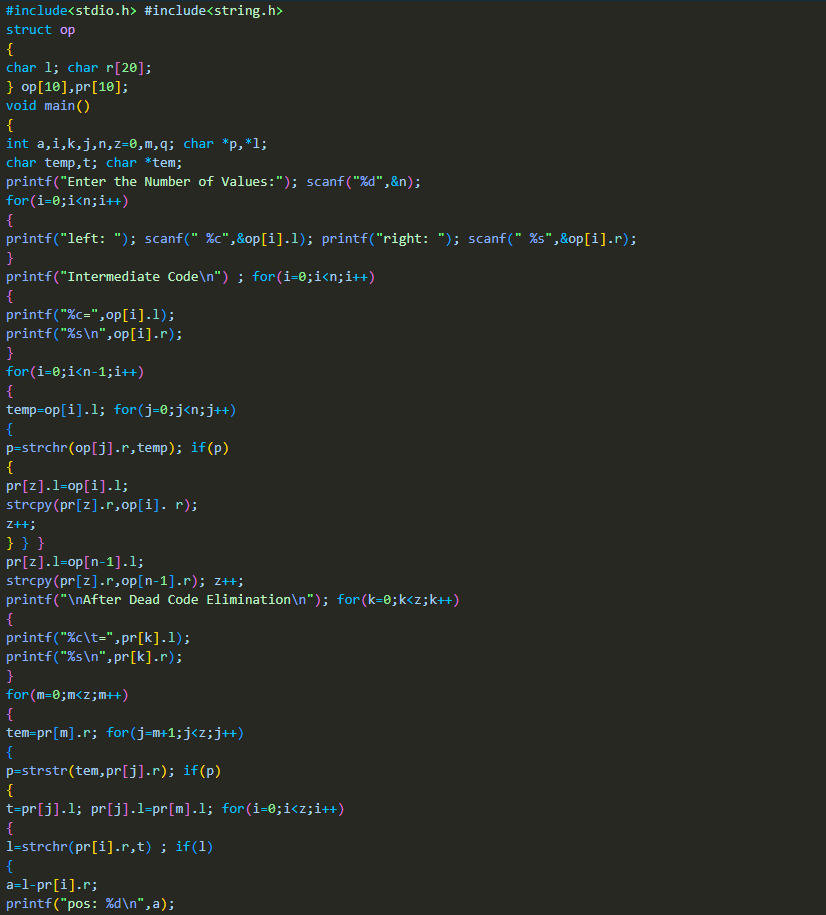
**Exercise 7**

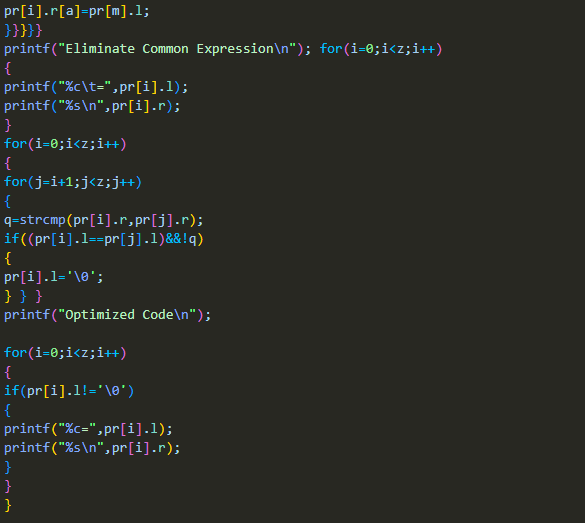
Aim: To implementation of Code Optimization Techniques

Algorithm:

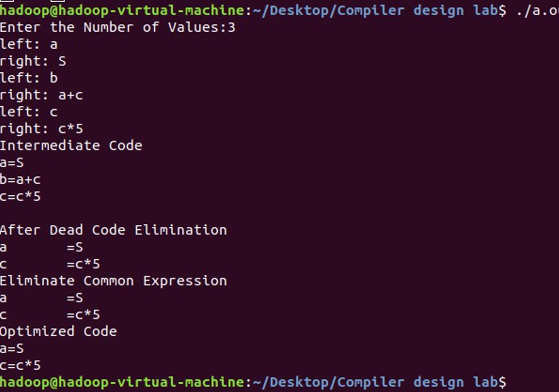
1. Start the program
2. Declare the variables and functions.
3. Enter the expression and state it in the variable a, b, c.
4. Calculate the variables b & c with ‘temp’ and store it in f1 and f2.
5. If(f1=null && f2=null) then expression could not be optimized.
6. Print the results.
7. Stop the program.

Code :





Output :



**Result:** Thus, the program to implement code optimization has been executed successfully.

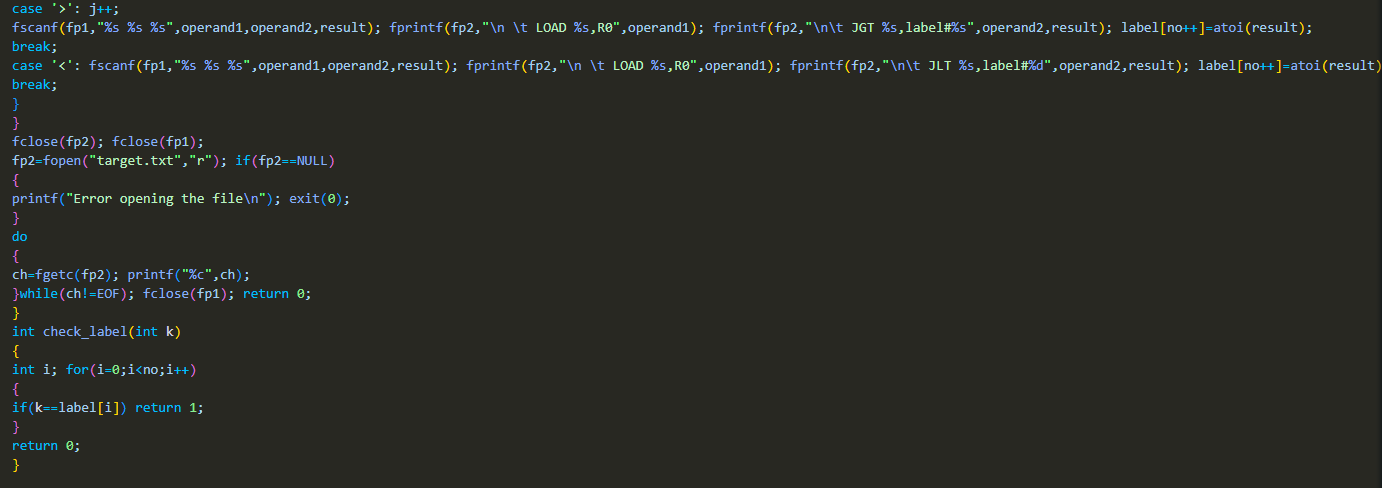
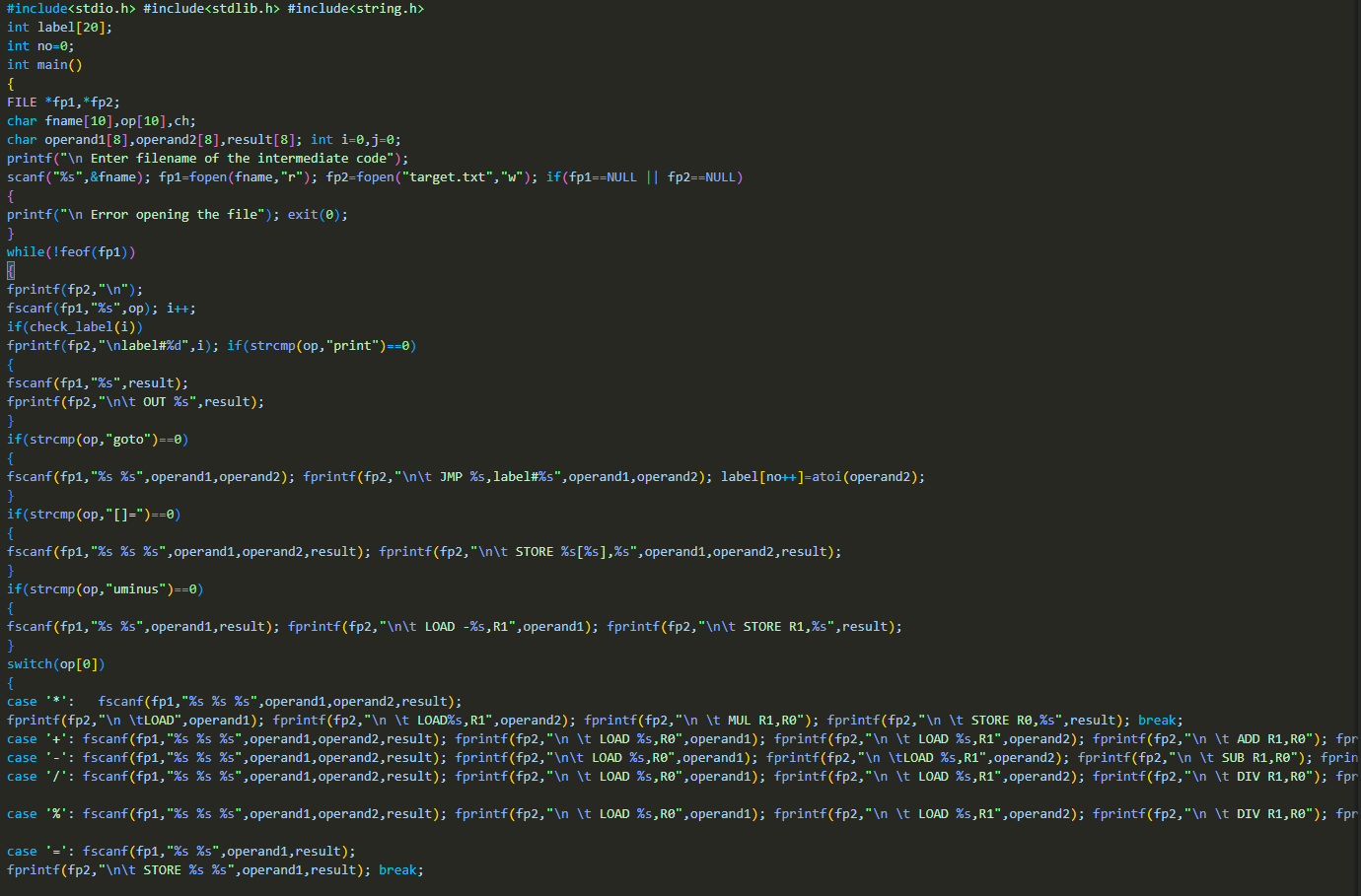
**Exercise 8**

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

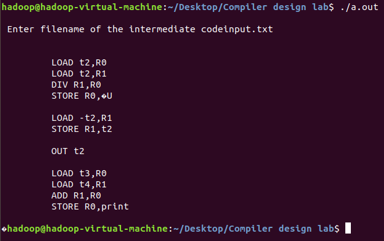
Algorithm:

1. Read input string
2. Consider each input string and convert it to machine code instructions using switch case
3. Load the input variables into new variables as operands and display them using “load”
4. With the help of arithmetic operation, we will display arithmetic operations like add, sub, div, mul for the respective operations in switch case
5. Generate 3 address code for each input variable.
6. If ‘=‘ is seen as arithmetic operation, then store the result in a variable and display it with “store”.
7. Repeat this for each line in the input string.
8. Display the output which gives a transformed input string of assembly language code.

Code :



Output :



**Result:** Thus, the program to implement target code generation has been successfully executed.