LAB RECORD

COMPILER DESIGN

19CSE401

Amrita School of Computing

Amrita Vishwa Vidyapeetham

Chennai – 601103, Tamil Nadu, India.

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

BONAFIDE CERTIFICATE

University Reg. No: CH.EN.U4CSE22158

This is to certify that this is a Bonafide record work done by Mr. **Vaka Tharun Kumar** studying B.Tech 4th year – Computer Science and engineering department

Internal Examiner1 **Internal Examiner2**

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

INDEX

EXPERIMENT NO: 1

DATE: 07-07-2025

TITLE OF THE PROGRAM: To implement Lexical Analyzer Using Lex Tool

AIM: To design and implement a Lexical Analyzer using the Lex tool that automatically identifies, categorizes, and displays various lexical components (tokens) such as keywords, identifiers, constants, operators, strings, comments, and preprocessor directives from a C program source file. The objective is to demonstrate the working of the first phase of a compiler — Lexical Analysis — by using regular expressions to define token patterns and by generating an efficient C-based lexical analyzer that scans the source code and classifies tokens appropriately, ignoring comments and whitespace, and handling unrecognized symbols gracefully.

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

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:

```
%{

#include <stdio.h>

#include <stdlib.h>


int COMMENT = 0;

%}


%option noyywrap identifier [a-

zA-Z_][a-zA-Z0-9_]* number [0-

9]+

string \".*\"
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

```
%%


"/*"                { COMMENT = 1; printf("\n%s begins a COMMENT", yytext); }

"*/"                { COMMENT = 0; printf("\n%s ends a COMMENT", yytext); }


#.*                 { if (!COMMENT) printf("\n%s is a PREPROCESSOR DIRECTIVE", yytext); }


"int" |

"float" |

"char" |

"double" |

"while" |

"for" |

"struct" |

"typedef" |

"do" |

"if" |

"break" |

"continue" |

"void" |

"switch" |

"return" |

"else" |

"goto"              { if (!COMMENT) printf("\n%s is a KEYWORD", yytext); }


{identifier}\(      { if (!COMMENT) printf("\nFUNCTION CALL or DECLARATION: %s", yytext); }


\{                  { if (!COMMENT) printf("\nBLOCK BEGINS"); }
```

```
\}                  { if (!COMMENT) printf("\nBLOCK ENDS"); }

{identifier}(\[[0-9]*\])? { if (!COMMENT) printf("\n%s is an IDENTIFIER", yytext); }

{string}            { if (!COMMENT) printf("\n%s is a STRING", yytext); }

{number}            { if (!COMMENT) printf("\n%s is a NUMBER", yytext); }

=                   { if (!COMMENT) printf("\n%s is an ASSIGNMENT OPERATOR", yytext); }

"<=" |
">=" |
"<"  |
"==" |
">"                 { if (!COMMENT) printf("\n%s is a RELATIONAL OPERATOR", yytext); }

[ \t\n]+            ;   // Ignore whitespace

.                   { if (!COMMENT) printf("\nUnrecognized: %s", yytext); }

%%

int main() {
   FILE *file = fopen("var.c", "r");
   if (!file) {
       fprintf(stderr, "Could not open file.\n");
exit(1);
   }
   yyin = file;
yylex();
printf("\n");
fclose(file);
return 0;
}
```

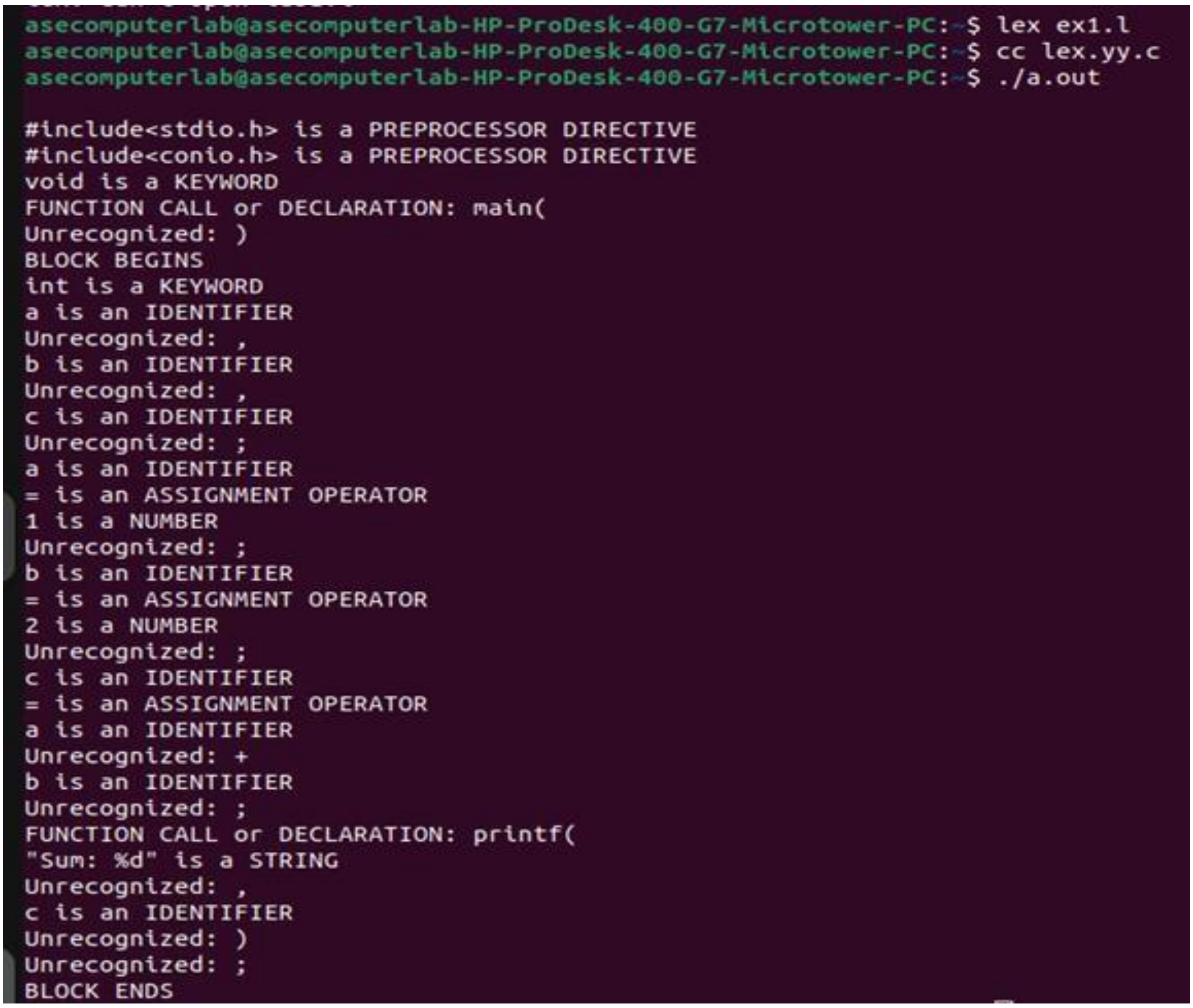
Variable.c:

```
#include<stdio.h>
#include<conio.h>
void main() {
   int a, b, c;     a = 1;
b = 2;     c = a + b;
printf("Sum: %d", c);
}
```

OUTPUT:

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ lex ex1.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ cc lex.yy.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out

#include<stdio.h> is a PREPROCESSOR DIRECTIVE
#include<conio.h> is a PREPROCESSOR DIRECTIVE
void is a KEYWORD
FUNCTION CALL or DECLARATION: main(
Unrecognized: )
BLOCK BEGINS
int is a KEYWORD
a is an IDENTIFIER
Unrecognized: ,
b is an IDENTIFIER
Unrecognized: ,
c is an IDENTIFIER
Unrecognized: ;
a is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
1 is a NUMBER
Unrecognized: ;
b is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
2 is a NUMBER
Unrecognized: ;
c is an IDENTIFIER
= is an ASSIGNMENT OPERATOR
a is an IDENTIFIER
Unrecognized: +
b is an IDENTIFIER
Unrecognized: ;
FUNCTION CALL or DECLARATION: printf(
"Sum: %d" is a STRING
Unrecognized: ,
c is an IDENTIFIER
Unrecognized: )
Unrecognized: ;
BLOCK ENDS
```

RESULT: The code has been executed and output displayed successfully.

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

EXPERIMENT NO:

07-07-2025

TITLE OF THE PROGRAM:

1(a)

DATE:

To find the number of vowels and consonant in the sentence

AIM: To write a Lexical Analyzer using LEX that counts the number of vowels and consonants in a given input string.

ALGORITHM:

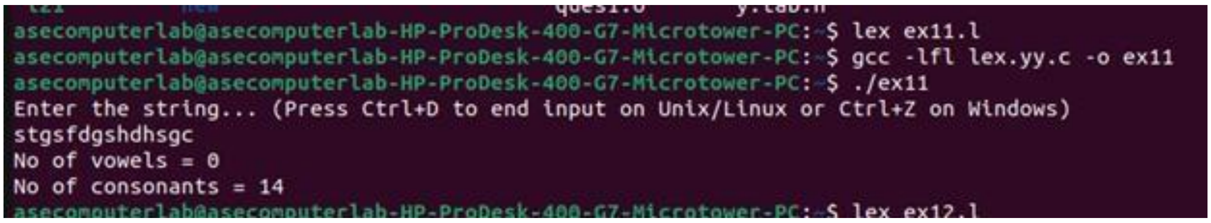
1. Start the program.
2. Initialize two variables: vowels = 0 and cons = 0.
3. Define regular expressions for:
4. Vowels [aeiouAEIOU]
5. Alphabet letters [a-zA-Z]
6. For each character in the input string:
7. If it matches a vowel, increment the vowels counter.
8. If it matches an alphabet letter, increment the cons counter.
9. Ignore all other characters.
10. Subtract the number of vowels from cons to get the count of consonants.
11. Display the total number of vowels and consonants.
12. End the program.

CODE:

```
%{
#include <stdio.h>


int vowels = 0;
int cons = 0;
%}


%%
[aeiouAEIOU]     { vowels++; }
[a-zA-Z]         { cons++; }
```

```
.|\n          { /* ignore other characters */ }
%%


int yywrap() {
   return 1;
}


int main() {
   printf("Enter the string... (Press Ctrl+D to end input on Unix/Linux or Ctrl+Z on
Windows)\n");
   yylex();
   printf("No of vowels = %d\n", vowels);    printf("No of consonants = %d\n", cons -
vowels); // subtract vowels to get consonants     return 0;
}
```

OUTPUT:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ lex ex11.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc -lfl lex.yy.c -o ex11
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./ex11
Enter the string... (Press Ctrl+D to end input on Unix/Linux or Ctrl+Z on Windows)
stgsfdgshdhsgc
No of vowels = 0
No of consonants = 14
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ lex ex12.l
```

RESULT: We have successfully implemented an lexical analyser for counting vowels and consonants

1(b)

To find if the given input is a digit or not

AIM: To write a **Lexical Analyzer** using LEX that checks whether the given input is a **digit** or **not a digit**, and displays the corresponding message.

ALGORITHM:

1. Start the program.

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

EXPERIMENT NO:

DATE: 07-07-2025

TITLE OF THE PROGRAM:

2. Include the necessary header files (stdio.h and stdlib.h).
3. Define Lex rules for identifying input:
   - If the input matches digits (0-9), classify it as a **digit**.
   - If the input contains any non-digit characters or alphabets, classify it as **not a digit**.
4. Ignore any other characters.
5. Use the yylex() function to start lexical analysis.
6. Display the result: either "digit" or "not a digit" based on the input.
7. End the program.

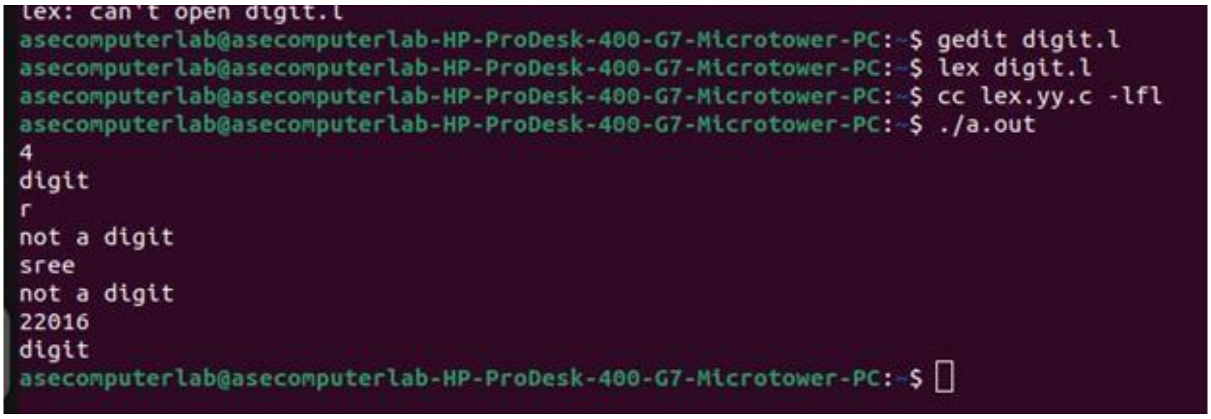
CODE:

```
/* Check whether input is digit or not. */
%{
#include<stdio.h>
#include<stdlib.h>
%}
/* Rule Section */
%%
^[0-9]*  printf("digit");
^[^0-9]|[0-9]*[a-zA-Z]  printf("not a digit");
. ;
%%
int main()
{
      // The function that starts the
analysis     yylex();        return 0;
}
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

OUTPUT:



```
lex: can't open digit.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit digit.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ lex digit.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ cc lex.yy.c -lfl
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
4
digit
r
not a digit
sree
not a digit
22016
digit
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

RESULT: We have successfully implemented an lexical analyser to find if the given input is an digit or not

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

EXPERIMENT NO:

DATE: 07-07-2025

TITLE OF THE PROGRAM:

1(c)

To find the number of words in the input

AIM: To write a **Lexical Analyzer** using LEX that counts the **number of words** in a given input text and displays the count for each line.

ALGORITHM:

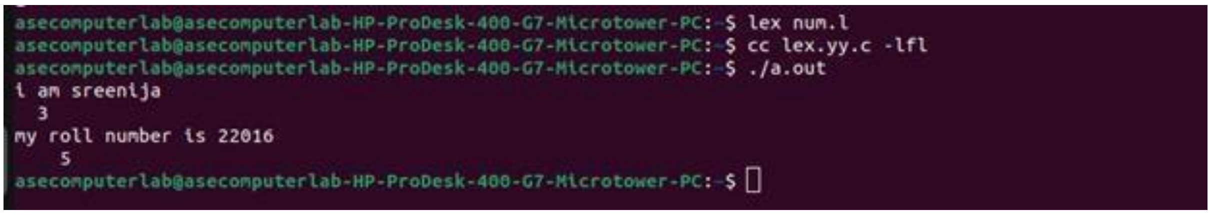
1. Start the program.
2. Include the necessary header files (stdio.h and string.h).
3. Initialize a counter variable i = 0 to keep track of the number of words.
4. Define Lex rules:
   - ([a-zA-Z0-9])* → Increment the word counter i for each word.
   - \n → Print the current word count i and reset i to 0 for the next line.
5. Use yylex() to start lexical analysis on the input text.
6. Continue processing until the end of input.
7. End the program.

CODE:

```
/*lex program to count number of words*/

%{

#include<stdio.h> #include<string.h>

int i = 0;

%}


/* Rules Section*/

%%
```

```
([a-zA-Z0-9])*    {i++;} /* Rule for counting
number of words*/

"\n" {printf("%d\n", i); i = 0;}
%%


int yywrap(void){}


int main()
{
   // The function that starts the analysis
yylex();


   return 0;
}
```

OUTPUT:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ lex num.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ cc lex.yy.c -lfl
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
i am sreenija
  3
my roll number is 22016
    5
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

RESULT: We have successfully implemented an lexical analyser for counting number of words
1(d)

To find if the given number is an odd or an even number

AIM: To write a **Lexical Analyzer** using LEX that reads an integer input and determines whether the number is **even** or **odd**, and displays the result accordingly.

ALGORITHM:

1. Start the program.
2. Include the necessary header file (stdio.h).
3. Declare an integer variable i to store the input number.

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

EXPERIMENT NO:

DATE: 07-07-2025

TITLE OF THE PROGRAM:

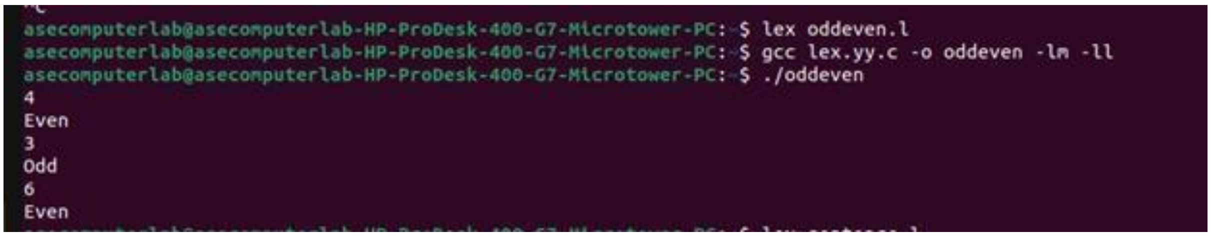
4. Define a Lex rule:
    - [0-9]+ → Convert the matched string to an integer using atoi(yytext).
    - Check if the number is divisible by 2:
        - If divisible, print "Even".
        - Otherwise, print "Odd".
5. Use yylex() to start lexical analysis and process the input.
6. Continue processing until the end of input.
7. End the program.

CODE:

```
%{
#include<stdio.h>
int i;
%}

%%

[0-9]+    {i=atoi(yytext);
if(i%2==0)
printf("Even");         else
printf("Odd");}
%%

int yywrap(){}
```

```
/* Driver code */

int main()

{


   yylex();

return 0;

}
```

OUTPUT:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ lex oddeven.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc lex.yy.c -o oddeven -lm -ll
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./oddeven
4
Even
3
Odd
6
Even
```

RESULT: We have successfully implemented an lexical analyser for finding odd or even number

1(e)

To find if the given input is a simple or a complex sentence

AIM: To write a Lexical Analyzer using LEX that determines whether a given sentence is simple or compound by checking for the presence of conjunctions like and, or, and but.

ALGORITHM:

1. Start the program.
2. Include the necessary header file (stdio.h).
3. Initialize a flag variable is_simple = 1 to indicate a simple sentence.
4. Define Lex rules:
   - [\t\n]+[aA][nN][dD][\t\n]+ → Set is_simple = 0 if the word and is found.
   - [\t\n]+[oO][rR][\t\n]+ → Set is_simple = 0 if the word or is found. ○ [\t\n]+[bB][uU][tT][\t\n]+ → Set is_simple = 0 if the word but is found.

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

EXPERIMENT NO:

DATE: 07-07-2025

TITLE OF THE PROGRAM:

   - . → Ignore all other characters.

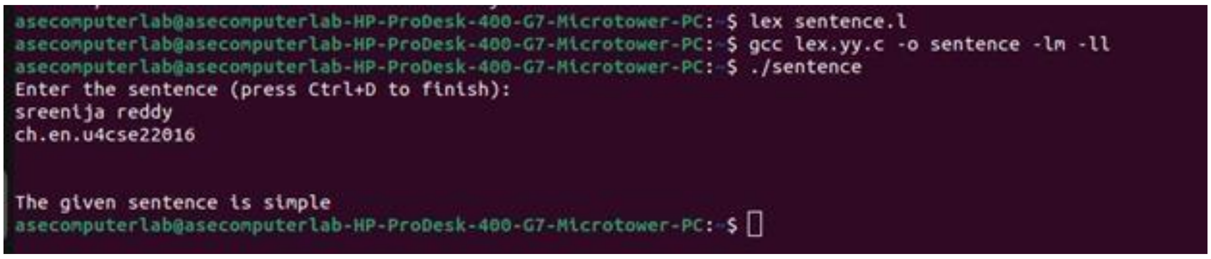
5. Use yylex() to scan the input sentence.
6. After scanning:
   - If is_simple == 1, print "The given sentence is simple".
   - Otherwise, print "The given sentence is compound".
7. End the program.

CODE:

```
%{
#include <stdio.h>
int is_simple = 1;
%}

%%
[ \t\n]+[aA][nN][dD][ \t\n]+    { is_simple = 0; }
[ \t\n]+[oO][rR][ \t\n]+        { is_simple = 0; }
[ \t\n]+[bB][uU][tT][ \t\n]+    { is_simple = 0; }
.                               { /* ignore other chars */ }
%%

int yywrap() {
return 1;
}

int main() {     printf("Enter the sentence (press Ctrl+D
to finish):\n");     yylex();
```

```
    if (is_simple == 1) {         printf("The given
sentence is simple\n");
    } else {
        printf("The given sentence is compound\n");
    }

    return 0;
}
```

OUTPUT:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ lex sentence.l
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc lex.yy.c -o sentence -lm -ll
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./sentence
Enter the sentence (press Ctrl+D to finish):
sreenija reddy
ch.en.u4cse22016


The given sentence is simple
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$
```

RESULT: We have successfully implemented an lexical analyser for finding simple or a complex sentence

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

EXPERIMENT NO:

DATE:

:

2(a)

21-07-2025

TITLE OF THE PROGRAM To implement eliminate left recursion and left factoring from the given grammar using C program.

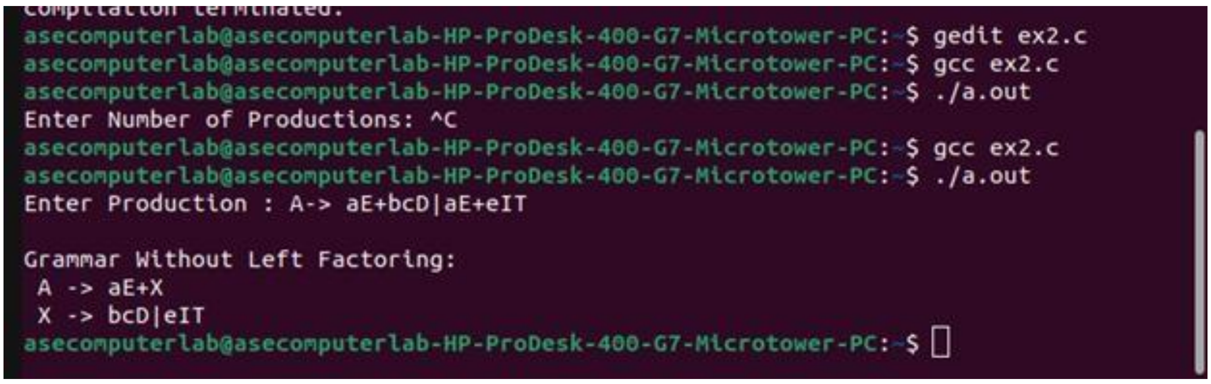
AIM: To implement Left Factoring of a given grammar production using C programming, by identifying the common prefix in the alternatives of a production and rewriting the grammar to remove the left recursion or ambiguity, producing a modified grammar and a new non-terminal for the factored part.

ALGORITHM:

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

```
#include<stdio.h>

#include<string.h>

int main()

{

   char

gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];     int

i,j=0,k=0,l=0,pos;     printf("Enter Production : A->");     gets(gram);

for(i=0;gram[i]!='|';i++,j++)         part1[j]=gram[i];     part1[j]='\0';

for(j=++i,i=0;gram[j]!='\0';j++,i++)         part2[i]=gram[j];     part2[i]='\0';

for(i=0;i<strlen(part1)||i<strlen(part2);i++){         if(part1[i]==part2[i]){

modifiedGram[k]=part1[i];

          k++;

pos=i+1;

       }
   }
   for(i=pos,j=0;part1[i]!='\0';i++,j++){
newGram[j]=part1[i];
   }
   newGram[j++]='|';
for(i=pos;part2[i]!='\0';i++,j++){
newGram[j]=part2[i];
   }
   modifiedGram[k]='X';
modifiedGram[++k]='\0';     newGram[j]='\0';
printf("\nGrammar Without Left Factoring : : \n");
printf(" A->%s",modifiedGram);     printf("\n X-
>%s\n",newGram);
}
```

OUTPUT:



```
compilation terminated.
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit ex2.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc ex2.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
Enter Number of Productions: ^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc ex2.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
Enter Production : A-> aE+bcD|aE+eIT

Grammar Without Left Factoring:
 A -> aE+X
 X -> bcD|eIT
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

RESULT: We have implemented left factoring successfully

2(b)

21-07-2025

TITLE OF THE PROGRAM To implement left recursion using C

AIM: To identify and eliminate immediate left recursion from a given set of grammar productions using C programming, by detecting productions where the non-terminal recurs

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

EXPERIMENT NO:

DATE:

:

at the beginning of its own production and rewriting them to produce an equivalent grammar without left recursion.

ALGORITHM:

1. Start the process 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:

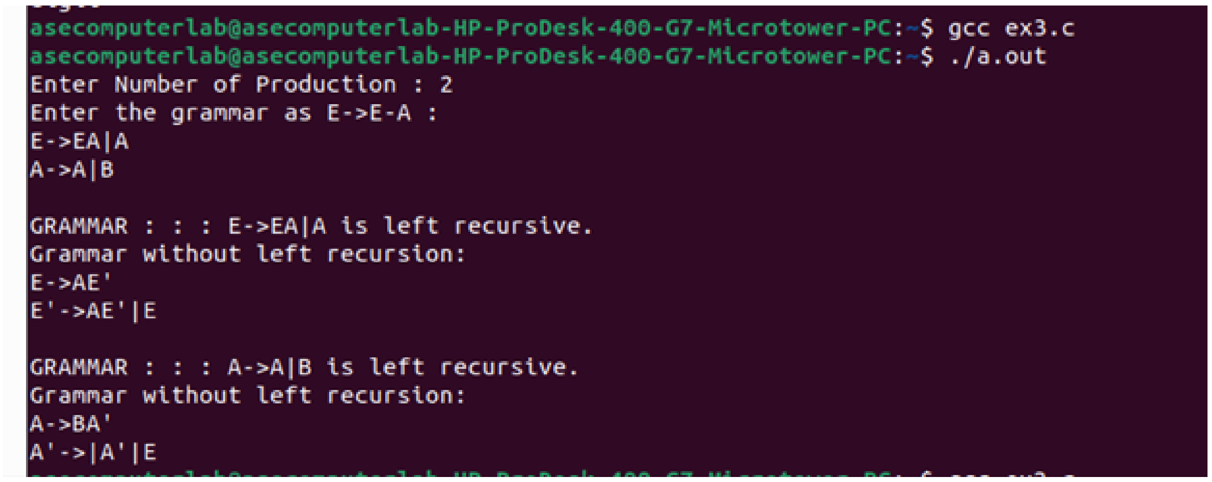
```
#include<stdio.h>

#include<string.h>   #define SIZE 10    int main () {

char non_terminal;       char beta,alpha;       int

num;       char production[10][SIZE];       int

index=3; /* starting of the string following "->" */

printf("Enter Number of Production : ");

scanf("%d",&num);       printf("Enter the grammar

as E->E-A :\n");       for(int i=0;i<num;i++){

scanf("%s",production[i]);

    }

    for(int i=0;i<num;i++){

printf("\nGRAMMAR : : : %s",production[i]);

non_terminal=production[i][0];

if(non_terminal==production[i][index]) {

alpha=production[i][index+1];

printf(" is left recursive.\n");

while(production[i][index]!=0 &&
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

```
production[i][index]!='|')                    index++;
if(production[i][index]!=0) {
beta=production[i][index+1];
printf("Grammar without left recursion:\n");
printf("%c-
>%c%c\'",non_terminal,beta,non_terminal);
printf("\n%c\'-
>%c%c\'|E\n",non_terminal,alpha,non_termin
al);  )              else
                   printf(" can't be
reduced\n;}}          else
printf(" is not left recursive.\n");
index=3;
     }
 }
```

OUTPUT



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc ex3.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
Enter Number of Production : 2
Enter the grammar as E->E-A :
E->EA|A
A->A|B

GRAMMAR : : : E->EA|A is left recursive.
Grammar without left recursion:
E->AE'
E'->AE'|E

GRAMMAR : : : A->A|B is left recursive.
Grammar without left recursion:
A->BA'
A'->|A'|E
```

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

3

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

EXPERIMENT NO:

DATE:

:

11-09-2025

TITLE OF THE PROGRAM To implement LL(1) parsing using C program.

AIM: To implement a Predictive Parser for a given grammar using C programming, by constructing a parsing table and simulating the parsing process with a stack, in order to check whether an input string is accepted or rejected according to the grammar rules.

ALGORITHM:

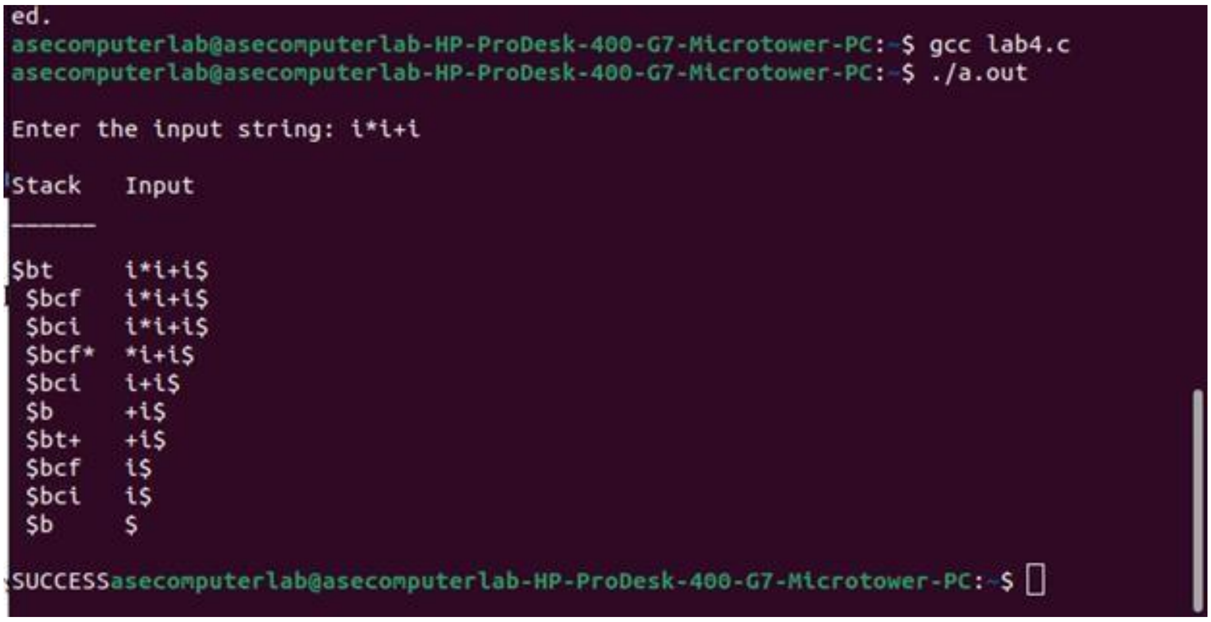
1. Read the input string.
2. Using the predictive parsing table, parse the given input using the stack.
3. If stack[i] matches with the input token, pop the token; else shift it and repeat the process until it reaches $.

CODE:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

char s[20], stack[20];

int main() {     char

m[5][6][3] = {

        {"tb", "",   "",   "tb", "",   ""},

        {"",  "+tb", "",   "",  "n",  "n"},

        {"fc", "",   "",   "fc", "",   ""},

        {"",  "n",  "*fc", "n",  "n",  "n"},

        {"i", "",    "",   "(e)", "",  ""}

    };

    int size[5][6] = {

{2, 0, 0, 2, 0, 0},

        {0, 3, 0, 0, 1, 1},

        {2, 0, 0, 2, 0, 0},
        {0, 1, 3, 0, 1, 1},
        {1, 0, 0, 3, 0, 0}
    };
    int i, j, k, n, str1, str2;
printf("\nEnter the input string: ");
scanf("%s", s);     strcat(s, "$");     n
= strlen(s);     stack[0] = '$';
stack[1] = 'e';     i = 1;     j = 0;
    printf("\nStack\tInput\n");
printf("______\n\n");     while
((stack[i] != '$') && (s[j] != '$')) {        if
(stack[i] == s[j]) {
            i--;
j++;
        }

        switch (stack[i]) {
case 'e': str1 = 0; break;
case 'b': str1 = 1; break;
case 't': str1 = 2; break;
case 'c': str1 = 3; break;
case 'f': str1 = 4; break;
        }

        switch (s[j]) {            case
'i': str2 = 0; break;
case '+': str2 = 1; break;
```

```
            case '*': str2 = 2; break;
case '(': str2 = 3; break;
case ')': str2 = 4; break;
case '$': str2 = 5; break;
        }

        if (m[str1][str2][0] == '\0') {
printf("\nERROR");
            exit(0);
        } else if (m[str1][str2][0] == 'n') {
            i--;
        } else if (m[str1][str2][0] == 'i') {
stack[i] = 'i';         } else {             for (k =
size[str1][str2] - 1; k >= 0; k--) {
stack[i] = m[str1][str2][k];

i++;
}            i--;
        }

        for (k = 0; k <= i; k++)
printf("%c", stack[k]);
printf("\t");         for (k = j;
k <= n; k++)
printf("%c", s[k]);
printf(" \n ");
    }
    printf("\nSUCCESS");
return 0;
}
```

OUTPUT:



```
ed.
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc lab4.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out

Enter the input string: i*i+i

Stack   Input
------

$bt     i*i+i$
 $bcf   i*i+i$
 $bci   i*i+i$
 $bcf*  *i+i$
 $bci   i+i$
 $b     +i$
 $bt+   +i$
 $bcf   i$
 $bci   i$
 $b     $

SUCCESSasecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

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

EXPERIMENT NO: 4

DATE: 25-09-2025

TITLE OF THE PROGRAM: To write a program in YACC for parser generation.

AIM: To implement a calculator using Bison and Lex that can evaluate arithmetic expressions involving addition, subtraction, multiplication, division, and unary minus, by defining grammar rules, operator precedence, and associativity, and computing the result of input expressions entered by the user.

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 the program.
3. Define the precedence and associativity of various operators like +, -, *, /, etc.
4. Write code for saving the answer into memory and displaying the result on the screen.

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

5. Write code 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:

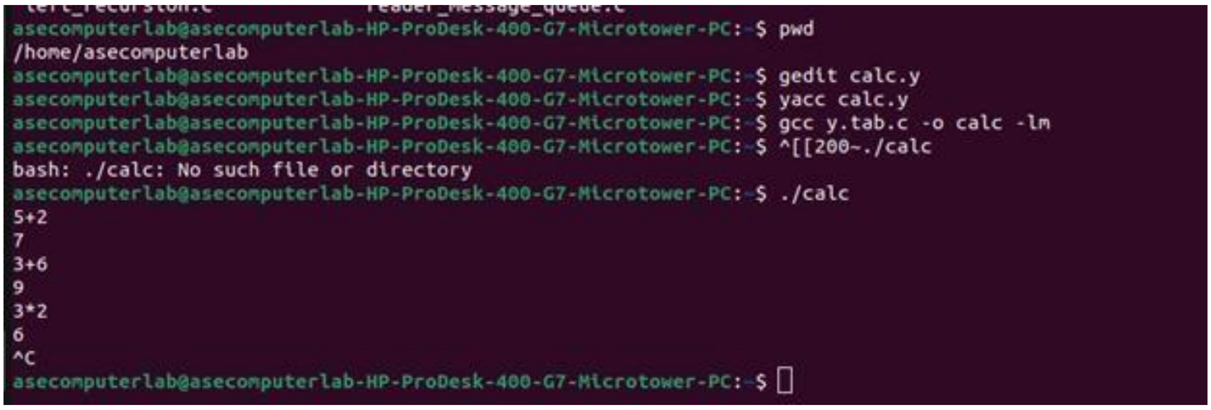
```
%{
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int yylex(); int
yyerror(const char *s);

#define YYSTYPE double
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%

lines:    lines expr '\n'    {
printf("%g\n", $2); }
  | lines '\n'
  | /* empty */
;
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

```
expr:
   expr '+' expr        { $$ = $1 + $3;
}   | expr '-' expr        { $$ = $1 -
$3; }
  | expr '*' expr        { $$ = $1 * $3; }
  | expr '/' expr        { $$ = $1 / $3; }
  | '-' expr %prec UMINUS { $$ = -$2; }
  | '(' expr ')'         { $$ = $2; }
  | NUMBER               { $$ = $1; };
%%
int yylex() {     int c;     while ((c =
getchar()) == ' ' || c == '\t');     if (c ==
EOF) return 0;     if (isdigit(c) || c == '.')
{        ungetc(c, stdin);        if
(scanf("%lf", &yylval) != 1) {
fprintf(stderr, "Invalid number\n");
            exit(1);
        }
        return NUMBER;
   }
   return c;
}
int yyerror(const char *s) {
fprintf(stderr, "Error: %s\n", s);
return 1;
}
int main() {
return yyparse();
}
int yywrap() {
return 1;
}
```

OUTPUT:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ pwd
/home/asecomputerlab
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit calc.y
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ yacc calc.y
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc y.tab.c -o calc -lm
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ^[[200~./calc
bash: ./calc: No such file or directory
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./calc
5+2
7
3+6
9
3*2
6
^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

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

EXPERIMENT NO: 5

DATE: 01-09-2025

TITLE OF THE PROGRAM: To implement symbol table

AIM: To implement a symbol table for identifiers and operators using C programming, by reading an input expression, allocating memory dynamically for each symbol, storing their addresses, and displaying the symbol along with its memory address and type.

ALGORITHM:

1. Start the program.
2. Get the input from the user with the terminating symbol $.
3. Allocate memory for the variable using dynamic memory allocation function.
4. If the next character of the symbol is an operator, then allocate memory.
5. While reading, insert the input symbol into the symbol table along with its memory address.
6. Repeat the steps until $ is reached.

NAME: Vaka Tharun Kumar

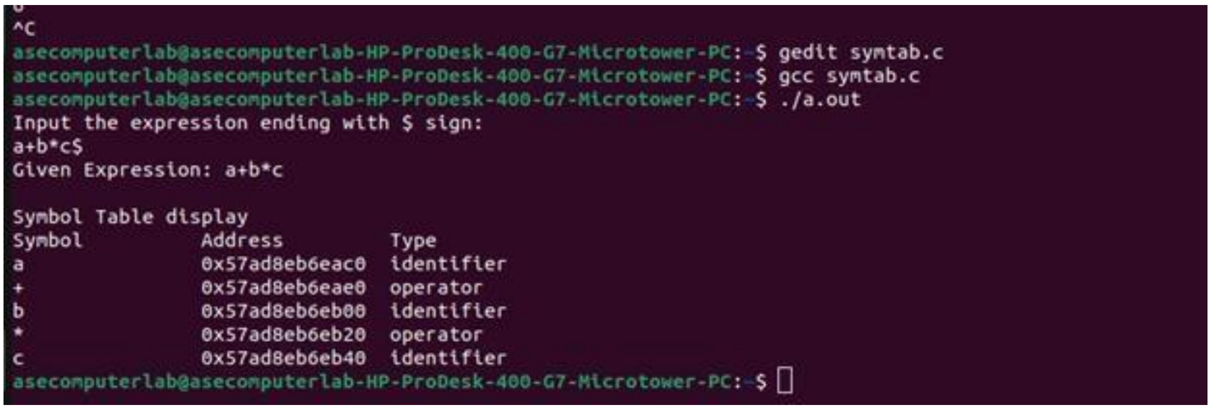
ROLL NO: CH.EN.U4CSE22158

7. To access a variable, enter the variable to be searched, and check the symbol table for the corresponding variable; display the variable along with its address as the result.
8. Stop the program.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h> int

main() {     int x = 0, n,

i = 0, j = 0;

   void *mypointer, *T4Tutorials_address[5];     char ch,

T4Tutorials_Array2[15], T4Tutorials_Array3[15], c;

printf("Input the expression ending with $ sign:\n");

while ((c = getchar()) != '$' && i < 14) {

T4Tutorials_Array2[i] = c;

       i++;

   }

   T4Tutorials_Array2[i] = '\0'; // Null-terminate the

string     n = i - 1;


   // Display input expression

printf("Given Expression: ");     for (i =

0; i <= n; i++) {        printf("%c",

T4Tutorials_Array2[i]);

   }

   printf("\n\nSymbol Table display\n");

printf("Symbol\t\tAddress\t\tType\n");

while (j <= n) {        c =
T4Tutorials_Array2[j];        if (isalpha(c))
{
          // Allocate memory for identifier
mypointer = malloc(sizeof(char));            if
(mypointer == NULL) {
printf("Memory allocation failed\n");
              return 1;
          }
          T4Tutorials_address[x] = mypointer;
T4Tutorials_Array3[x] = c;
printf("%c\t\t%p\tidentifier\n", c, mypointer);
x++;
       } else if (c == '+' || c == '-' || c == '*' || c ==
'=') {            // Allocate memory for operator
mypointer = malloc(sizeof(char));            if
(mypointer == NULL) {               printf("Memory
allocation failed\n");
              return 1;
          }
          T4Tutorials_address[x] = mypointer;
T4Tutorials_Array3[x] = c;
printf("%c\t\t%p\toperator\n", c, mypointer);
x++;
       }
j++;
    }
   // Free allocated memory
   for (i = 0; i < x; i++) {
free(T4Tutorials_address[i]);
   }
   return 0;
}
```

OUTPUT:



```
^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit symtab.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc symtab.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
Input the expression ending with $ sign:
a+b*c$
Given Expression: a+b*c

Symbol Table display
Symbol          Address         Type
a               0x57ad8eb6eac0  identifier
+               0x57ad8eb6eae0  operator
b               0x57ad8eb6eb00  identifier
*               0x57ad8eb6eb20  operator
c               0x57ad8eb6eb40  identifier
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

RESULT: Thus the program to implement symbol table has been executed successfully
EXPERIMENT NO: 6

DATE: 08-09-2025

TITLE OF THE PROGRAM: To implementation of intermediate code generation.

AIM: To implement **Intermediate Code Generation** for arithmetic expressions using C programming, by analyzing an input expression, identifying operators according to their precedence, and generating three-address code (TAC) that represents the expression in an intermediate form suitable for further compilation phases.

ALGORITHM:

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

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

```
int i = 1, j = 0, no = 0, tmpch = 90; // tmpch starts at 'Z'
char str[100], left[15], right[15];


struct exp {
int     pos;
char op;
} k[15];


// Function
declarations void
findopr(); void
explore(); void
fleft(int); void
fright(int);
int main() {    printf("\t\tINTERMEDIATE CODE
GENERATION\n\n");    printf("Enter the Expression : ");
scanf("%s", str);


   printf("The intermediate code:\n");


   findopr();
explore();


   return 0;
}


void findopr() {
   // Find operators in order of precedence (lowest to highest)
   // Store their positions and operator in array k


   for (i = 0; str[i] != '\0';
i++)        if (str[i] == ':') {
k[j].pos = i;
k[j++].op = ':';
       }

   for (i = 0; str[i] != '\0';
i++)        if (str[i] == '/') {
k[j].pos = i;
k[j++].op = '/';
       }
   for (i = 0; str[i] != '\0';
i++)        if (str[i] == '*') {
k[j].pos = i;
k[j++].op = '*';
       }

   for (i = 0; str[i] != '\0';
i++)        if (str[i] == '+') {
k[j].pos = i;
k[j++].op = '+';
       }

   for (i = 0; str[i] != '\0';
i++)        if (str[i] == '-') {
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

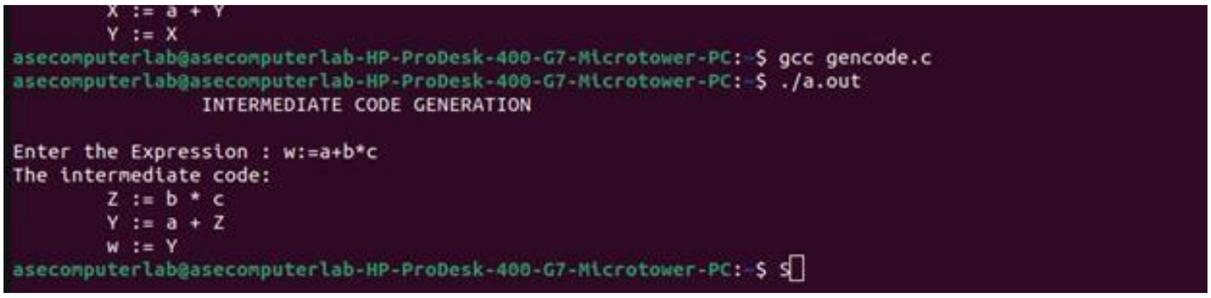
```
k[j].pos = i;
k[j++].op = '-';
        }
    // Mark end of operators array with a null char
k[j].op = '\0';
}

void explore() {
    i = 1;     while
(k[i].op != '\0') {
fleft(k[i].pos);
fright(k[i].pos);

        str[k[i].pos] = tmpch--;  // Replace operator position in str with a temp var like Z, Y, X...
printf("\t%c := %s %c %s\n", str[k[i].pos], left, k[i].op, right);

        i++;
    }

    // For last operator or expression
part     fright(-1);     if (no == 0) {
fleft(strlen(str));         printf("\t%s :=
%s\n", right, left);         exit(0);
    }

    printf("\t%s := %c\n", right, str[k[--i].pos]);
}
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

```
void fleft(int x) {
int w = 0, flag = 0;
x--;
   while (x != -1 && str[x] != '+' && str[x] != '*' && str[x] != '='
&&           str[x] != '\0' && str[x] != '-' && str[x] != '/' &&
str[x] != ':') {        if (str[x] != '$' && flag == 0) {
left[w++] = str[x];           left[w] = '\0';           str[x] = '$';
flag = 1;
       }
x--;
   }
}


void fright(int x) {     int w = 0, flag = 0;     x++;     while (x != -1
&& str[x] != '+' && str[x] != '*' && str[x] != '\0' &&
str[x] != '=' && str[x] != ':' && str[x] != '-' && str[x] != '/') {
if (str[x] != '$' && flag == 0) {           right[w++] = str[x];
right[w] = '\0';           str[x] = '$';           flag = 1;
       }
       x++;
   }
}
```

OUTPUT:



```
        X := a + Y
        Y := X
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc gencode.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
                INTERMEDIATE CODE GENERATION

Enter the Expression : w:=a+b*c
The intermediate code:
        Z := b * c
        Y := a + Z
        w := Y
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ $
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

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

EXPERIMENT NO: 7

DATE: 15-09-2025

TITLE OF THE PROGRAM: To implementation of Code Optimization Techniques

AIM: To implement **code optimization techniques** using C programming, specifically **Dead Code Elimination** and **Common Subexpression Elimination**, by analyzing a set of intermediate code instructions, removing unnecessary computations, and generating an optimized version of the code.

ALGORITHM:

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

CODE:

```
#include <stdio.h> #include <string.h>

struct op {     char l;     char r[20]; }

op[10], pr[10]; int main() {     int a, i, k,

j, n, z = 0, m, q;     char *p, *l;     char

temp, t;     char *tem;     printf("Enter

the Number of Values: ");

scanf("%d", &n);

  for (i = 0; i < n; i++) {
        printf("left: ");
        scanf(" %c", &op[i].l); // space before %c to ignore
newline        printf("right: ");        scanf(" %s", op[i].r);
    }
    printf("\nIntermediate Code:\n");
    for (i = 0; i < n; i++) {
printf("%c = %s\n", op[i].l, op[i].r);
    }
    for (i = 0; i < n - 1; i++) {
temp = op[i].l;        for (j = 0;
j < n; j++) {            p =
strchr(op[j].r, temp);
            if (p) {                pr[z].l
= op[i].l;
strcpy(pr[z].r, op[i].r);
z++;                break;
            }
        }
    }
    pr[z].l = op[n - 1].l;     strcpy(pr[z].r, op[n -
1].r);     z++;     printf("\nAfter Dead Code
Elimination:\n");     for (k = 0; k < z; k++) {
printf("%c = %s\n", pr[k].l, pr[k].r);
    }
   for (m = 0; m < z; m++) {
        tem = pr[m].r;        for (j = m
+ 1; j < z; j++) {            if
(strcmp(tem, pr[j].r) == 0) {
```

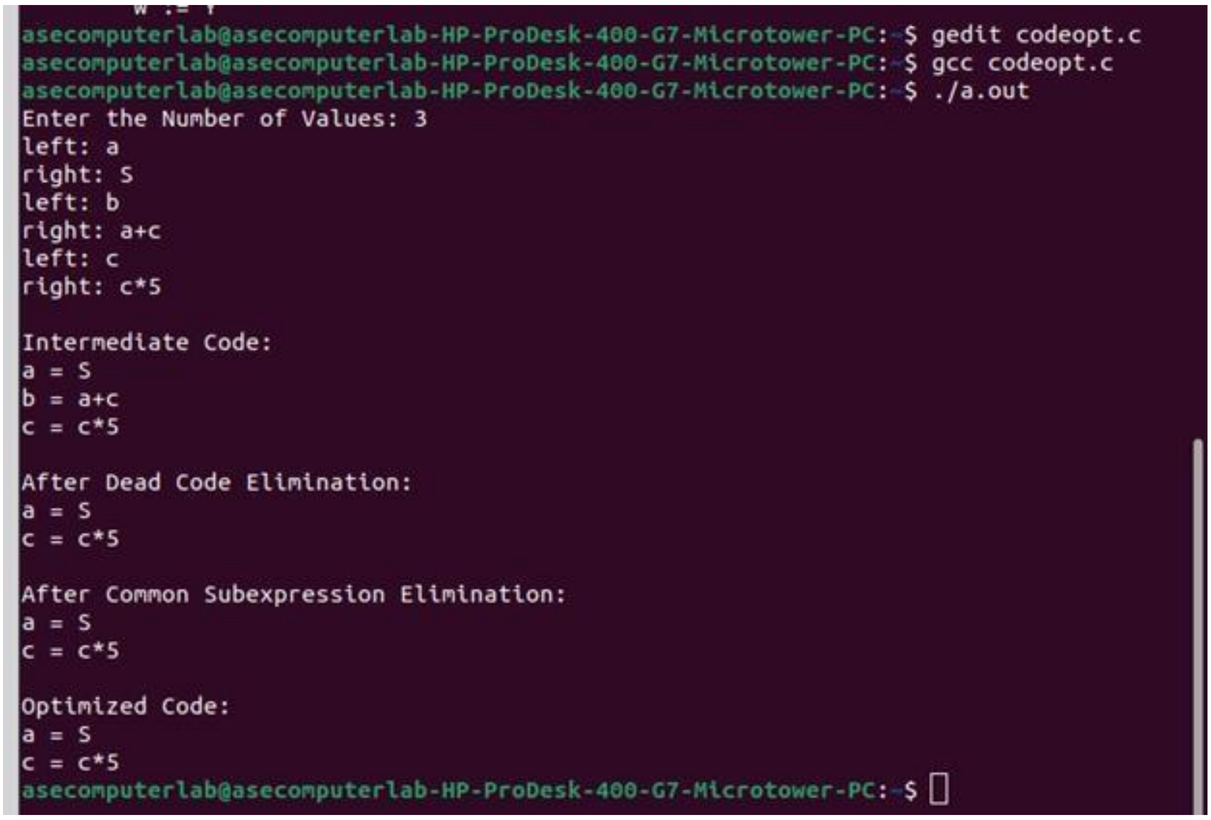
```
                t = pr[j].l;
pr[j].l = pr[m].l;                for
(i = 0; i < z; i++) {                    l
= strchr(pr[i].r, t);
if (l) {                        a = l -
pr[i].r;                        pr[i].r[a]
= pr[m].l;
                    }
                }
            }
        }
    }
    printf("\nAfter Common Subexpression Elimination:\n");
    for (i = 0; i < z; i++) {
printf("%c = %s\n", pr[i].l, pr[i].r);
    }
    // Remove duplicate assignments
    for (i = 0; i < z; i++) {         for (j = i +
1; j < z; j++) {             q =
strcmp(pr[i].r, pr[j].r);             if
((pr[i].l == pr[j].l) && q == 0) {
pr[i].l = '\0'; // mark as deleted
            }
        }
    }
   printf("\nOptimized Code:\n");
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

```
   for (i = 0; i < z; i++) {        if (pr[i].l !=
'\0') {           printf("%c = %s\n",
pr[i].l, pr[i].r);
       }
   }
   return 0;
}
```

OUTPUT:



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit codeopt.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc codeopt.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
Enter the Number of Values: 3
left: a
right: 5
left: b
right: a+c
left: c
right: c*5

Intermediate Code:
a = 5
b = a+c
c = c*5

After Dead Code Elimination:
a = 5
c = c*5

After Common Subexpression Elimination:
a = 5
c = c*5

Optimized Code:
a = 5
c = c*5
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

RESULT: Thus, the program to implement code optimization has been executed successfully
EXPERIMENT NO: 8

DATE: 22-09-2025

TITLE OF THE PROGRAM: To write a program that implements the target code generation

AIM: To implement a Target Code Generation program using C, which reads an intermediate code file, translates the instructions into corresponding target machine code or assembly-like instructions, handles operators, conditional and unconditional jumps, and generates an output file containing the target code.

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

ALGORITHM:

1. Read the 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. Perform arithmetic operations like ADD, SUB, MUL, DIV for the respective operators in the switch case.
5. Generate three-address code for each input variable.
6. If = is encountered as an arithmetic operation, store the result in a variable and display it using STORE.
7. Repeat the process for each line in the input string.
8. Display the output, which gives a transformed input string in assembly language code.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


int label[20];

int no = 0;


int check_label(int k) {

   for (int i = 0; i < no; i++) {

if (k == label[i]) return 1;

   }

   return 0;

}
```

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158

```
int main() {     FILE *fp1, *fp2;     char
fname[100], op[10], ch;     char
operand1[8], operand2[8], result[8];
    int i = 0;

    printf("Enter filename of the intermediate code: ");
scanf("%s", fname);

    fp1 = fopen(fname, "r");
fp2 = fopen("target.txt", "w");

    if (fp1 == NULL || fp2 == NULL)
{                printf("Error  opening
file.\n");        exit(1);
    }

    while (fscanf(fp1, "%s", op) != EOF) {
        i++;

        if (check_label(i)) {
fprintf(fp2, "\nlabel#%d:\n", i);
        }

        if (strcmp(op, "print") == 0) {
fscanf(fp1, "%s", result);            fprintf(fp2,
"\tOUT %s\n", result);        } else if
(strcmp(op, "goto") == 0) {
fscanf(fp1, "%s %s", operand1, operand2);
```

```
fprintf(fp2, "\tJMP %s, label#%s\n",
operand1, operand2);            label[no++] =
atoi(operand2);        } else if (strcmp(op,
"[]=") == 0) {           fscanf(fp1, "%s %s %s",
operand1, operand2, result);
fprintf(fp2, "\tSTORE %s[%s], %s\n",
operand1, operand2, result);
        } else if (strcmp(op, "uminus") == 0) {
fscanf(fp1, "%s %s", operand1, result);
fprintf(fp2, "\tLOAD -%s, R1\n", operand1);
fprintf(fp2, "\tSTORE R1, %s\n", result);
        } else {
            //              Handle
operators              switch
(op[0]) {              case '*':
                    fscanf(fp1, "%s %s %s", operand1, operand2,
result);                  fprintf(fp2, "\tLOAD %s, R0\n", operand1);
fprintf(fp2, "\tLOAD %s, R1\n", operand2);
fprintf(fp2, "\tMUL R1, R0\n");                  fprintf(fp2,
"\tSTORE R0, %s\n", result);                  break;              case
'+':
                    fscanf(fp1, "%s %s %s", operand1, operand2,
result);                  fprintf(fp2, "\tLOAD %s, R0\n", operand1);
fprintf(fp2, "\tLOAD %s, R1\n", operand2);
fprintf(fp2, "\tADD R1, R0\n");                  fprintf(fp2,
"\tSTORE R0, %s\n", result);                  break;

                case '-':
```

```
            fscanf(fp1, "%s %s %s", operand1, operand2,
result);            fprintf(fp2, "\tLOAD %s, R0\n", operand1);
fprintf(fp2, "\tLOAD %s, R1\n", operand2);
fprintf(fp2, "\tSUB R1, R0\n");            fprintf(fp2,
"\tSTORE R0, %s\n", result);            break;            case
'/':
            fscanf(fp1, "%s %s %s", operand1, operand2,
result);            fprintf(fp2, "\tLOAD %s, R0\n", operand1);
fprintf(fp2, "\tLOAD %s, R1\n", operand2);
fprintf(fp2, "\tDIV R1, R0\n");            fprintf(fp2, "\tSTORE
R0, %s\n", result);            break;            case '%':
            fscanf(fp1, "%s %s %s", operand1, operand2,
result);            fprintf(fp2, "\tLOAD %s, R0\n", operand1);
fprintf(fp2, "\tLOAD %s, R1\n", operand2);
fprintf(fp2, "\tMOD R1, R0\n");            fprintf(fp2,
"\tSTORE R0, %s\n", result);            break;            case
'=':
            fscanf(fp1, "%s %s", operand1, result);
fprintf(fp2, "\tSTORE %s, %s\n", operand1, result);
break;            case '>':
            fscanf(fp1, "%s %s %s", operand1, operand2,
result);            fprintf(fp2, "\tLOAD %s, R0\n", operand1);
fprintf(fp2, "\tJGT %s, label#%s\n", operand2, result);
label[no++] = atoi(result);            break;            case '<':
            fscanf(fp1, "%s %s %s", operand1, operand2,
result);            fprintf(fp2, "\tLOAD %s, R0\n", operand1);
fprintf(fp2, "\tJLT %s, label#%s\n", operand2, result);
label[no++] = atoi(result);            break;
```

```
            }
        }
    }

    fclose(fp1);
fclose(fp2);

    // Print the generated code     fp2 =
fopen("target.txt", "r");     if (fp2 ==
NULL) {         printf("Error opening
target.txt\n");         exit(1);
    }

    printf("\nGenerated Target Code:\n\n");
while ((ch = fgetc(fp2)) != EOF) {
putchar(ch);
    }

    fclose(fp2);
return 0;

}
```

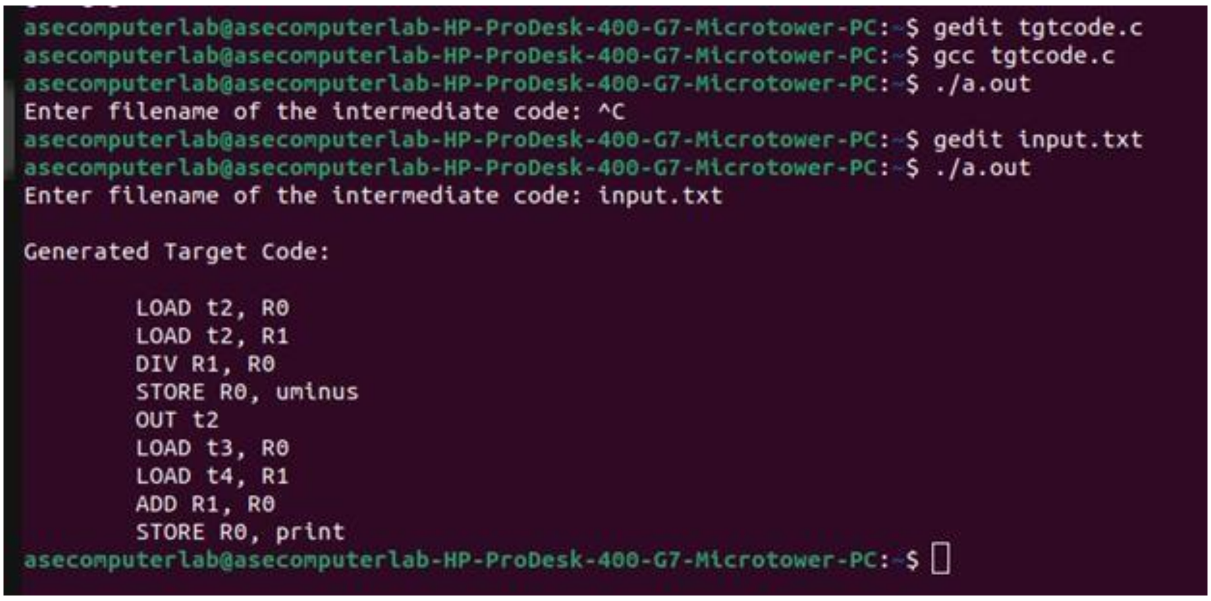
Input.txt:

```
/t3 t2 t2
uminus t2
t2 print t2
+t1 t3 t4
print t4
```

OUTPUT:

NAME: Vaka Tharun Kumar

ROLL NO: CH.EN.U4CSE22158



```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit tgtcode.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc tgtcode.c
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
Enter filename of the intermediate code: ^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit input.txt
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./a.out
Enter filename of the intermediate code: input.txt

Generated Target Code:

        LOAD t2, R0
        LOAD t2, R1
        DIV R1, R0
        STORE R0, uminus
        OUT t2
        LOAD t3, R0
        LOAD t4, R1
        ADD R1, R0
        STORE R0, print
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

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