COMPILER DESIGN LAB

19CSE401

INDEX

Date: 27/07/2022

Experiment No: 01

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:

```
%{

int COMMENT=0;

%}

identifier [a-zA-Z][a-zA-Z0-9]*

%%

#.* {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 {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}
\{  {if(!COMMENT)printf("\n BLOCK BEGINS");}
\}  {if(!COMMENT)printf("BLOCK ENDS ");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}
%%
int main(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
```

```
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}\
```

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:

```
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ lex lab1.l
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ cc lex.yy.c
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ ./a.out

#include<stdio.h> is a preprocessor directive

#include<conio.h> is a preprocessor directive

        void is a keyword
FUNCTION
        main(
        )


 BLOCK BEGINS

        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

 a IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 1 is a NUMBER ;

 b IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 2 is a NUMBER ;
```

```
 2 is a NUMBER ;

 c IDENTIFIER
          = is an ASSIGNMENT OPERATOR
 a IDENTIFIER+
 b IDENTIFIER;

FUNCTION
         printf(
          "Sum:%d" is a STRING,
 c IDENTIFIER
         )
;
BLOCK ENDS

hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$
```

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

Date: 10/08/2022

Experiment No: 02

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:

```
#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("\n A->%s",modifiedGram);
    printf("\n X->%s\n",newGram);
}
```

Output:

```
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ gcc lab2_factoring
.c
lab2_factoring.c: In function ‘main’:
lab2_factoring.c:8:8: warning: implicit declaration of function ‘gets’; did you
 mean ‘fgets’? [-Wimplicit-function-declaration]
        gets(gram);
        ^~~~
        fgets
/tmp/ccIFXWH1.o: In function `main':
lab2_factoring.c:(.text+0x5a): warning: the `gets' function is dangerous and sh
ould not be used.
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ ./a.out
Enter Production : A->aE+bcD|aE+eIT

 A->aE+X
 X->bcD|eIT
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$
```

Left recursion:

Aim: To implement left recursion using C.

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:

```
#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 && 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_terminal);
        }
        else
            printf(" can't be reduced\n");
    }
    else
        printf(" is not left recursive.\n");
     index=3;
  }
}
```

Output:

```
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ gedit lab2_recursi
on.c
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ gcc lab2_recursion
.c
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ ./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
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$
```

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

Date: 17/08/2022

Experiment No: 03

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:

```
#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"," a ","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("\n Enter 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");

printf("\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 'f': str1=4;
 break;
case 'c': str1=3;
 }
 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("\n SUCCESS");
   return 0;
```

}

Output:

```
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ gedit ll.c
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ gcc ll.c
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ ./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     $

 SUCCESShadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$
```

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

Date: 24/08/2022

Experiment No: 04

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:

```
%{

int yylex();

%}
```

```
%{
#define YYSTYPE double
#include <ctype.h>
#include <stdio.h>
%}
%token NUMBER
%left'+"-'
%left'*"/'
%right UMINUS
%%
lines:lines expr'\n'
{
printf("%g\n",$2);
}
|lines'\n'
|/*E*/
;
expr:expr'+'expr{$$=$1+$3;}
|expr'-'expr{$$=$1-$3;}
|expr'/'expr{$$=$1/$3;}
|'('expr')'{$$=$2;}
|'-'expr%prec UMINUS{$$=-$2;}
|NUMBER
;
%%
yylex()
{
int c;
while((c=getchar())==' ');
if((c=='.') || (isdigit(c)))
```

```
{
 ungetc(c,stdin);
 scanf("%lf",&yylval);
 return NUMBER;
}
return c;
}
int main()
{
yyparse();
return 1; }
int yyerror()
{
return 1;}
int yywrap()
{
return 1; }
```

Output:

```
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ yacc cal.y
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ gcc -o cal y.tab.c
y.tab.c: In function ‘yyparse’:
y.tab.c:1292:7: warning: implicit declaration of function ‘yyerror’; did you me
an ‘yyerrok’? [-Wimplicit-function-declaration]
       yyerror (YY_("syntax error"));
       ^~~~~~~
       yyerrok
cal.y: At top level:
cal.y:30:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 yylex()
 ^~~~~
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ ./cal
20+51
71
65-96
-31
245+658
903
210+65
275
```

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

Date: 21/09/2022

Experiment No: 05

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:

```
#include<stdio.h>
 #include<math.h>
 #include<string.h>
 #include<ctype.h>
 #include<stdlib.h>
 void main()
 {
  int x=0, n, i=0,j=0;
  void *mypointer,*T4Tutorials_address[5];
  char ch,T4Tutorials_Search,T4Tutorials_Array2[15],T4Tutorials_Array3[15],c;
  printf("Input the expression ending with $ sign:");
  while((c=getchar())!='$')
  {
   T4Tutorials_Array2[i]=c;
   i++;
  }
  n=i-1;
  printf("Given Expression:");
```

```
i=0;
while(i<=n)
{
 printf("%c",T4Tutorials_Array2[i]);
 i++;
}
printf("\n Symbol Table display\n");
printf("Symbol \t addr \t type");
while(j<=n)
{
 c=T4Tutorials_Array2[j];
 if(isalpha(toascii(c)))
 {
  mypointer=malloc(c);
  T4Tutorials_address[x]=mypointer;
  T4Tutorials_Array3[x]=c;
  printf("\n%c \t %d \t identifier\n",c,mypointer);
  x++;
  j++;
 }
 else
 {
  ch=c;
  if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
  {
   mypointer=malloc(ch);
   T4Tutorials_address[x]=mypointer;
   T4Tutorials_Array3[x]=ch;
   printf("\n %c \t %d \t operator\n",ch,mypointer);
   x++;
```

```
    j++;
    }}}}
```

Output:

```
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ ./a.out
Input the expression ending with $ sign:w=a+b*c$
Given Expression:w=a+b*c
 Symbol Table display
Symbol   addr    type
w        1840614016      identifier

 =       1840614144      operator

a        1840614224      identifier

 +       1840614336      operator

b        1840614400      identifier

 *       1840614512      operator

c        1840614576      identifier
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$
```

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

Date: 28/09/2022

Experiment No: 06

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:

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int i=1,j=0,no=0,tmpch=90;

char str[100],left[15],right[15];
```

```
void findopr();
void explore();
void fleft(int);
void fright(int);


 struct exp
 {
 int pos;
 char op;
 }k[15];
 void main()
 {
 printf("\t\tINTERMEDIATE CODE GENERATION\n\n");
 printf("Enter the Expression :");
 scanf("%s",str);
 printf("The intermediate code:\n");
 findopr();
 explore();
 }
 void findopr()
 {
 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]=='-')
{
 k[j].pos=i;
k[j++].op='-';
} }
void explore()
{
i=1;
while(k[i].op!='\0')
{
fleft(k[i].pos);
fright(k[i].pos);
str[k[i].pos]=tmpch--;
printf("\t%c := %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);
```

```
  printf("\n");
  i++;
  }
  fright(-1);
  if(no==0)
  {
  fleft(strlen(str));
  printf("\t%s := %s",right,left);
exit(0);
  }
  printf("\t%s := %c",right,str[k[--i].pos]);
}
  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:

```
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ gedit icg.c
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ gcc icg.c
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ ./a.out
                INTERMEDIATE CODE GENERATION

Enter the Expression :w:=a+b*c
The intermediate code:
        Z := b*c
        Y := a+Z
        w := Yhadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$
```

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

Date: 05/10/2022

Experiment No: 07

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:

```
#include<stdio.h>
#include<string.h>
struct op
{
char l;
char r[20];
}
op[10],pr[10];
void 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);
printf("right: ");
scanf(" %s",&op[i].r);
}
printf("Intermediate Code\n") ;
for(i=0;i<n;i++)
{
```

```
printf("%c=",op[i].l);
printf("%s\n",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++;
} } }
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\t=",pr[k].l);
printf("%s\n",pr[k].r);
}
for(m=0;m<z;m++)
{
tem=pr[m].r;
for(j=m+1;j<z;j++)
{
```

```
p=strstr(tem,pr[j].r);
if(p)
{
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;
printf("pos: %d\n",a);
pr[i].r[a]=pr[m].l;
}}}}}
printf("Eliminate Common Expression\n");
for(i=0;i<z;i++)
{
printf("%c\t=",pr[i].l);
printf("%s\n",pr[i].r);
}
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)
{
pr[i].l='\0';
} } }
printf("Optimized Code\n");
```

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

Output:

```
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ ./a.out
Enter the Number of Values:3
left: a
right: S
left: b
right: a+c
left: c
right: c*5
Intermediate Code
a=S
b=a+c
c=c*5

After Dead Code Elimination
a       =S
c       =c*5
Eliminate Common Expression
a       =S
c       =c*5
Optimized Code
a=S
c=c*5
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$
```

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

Date: 12/10/2022

Experiment No: 08

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:

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int label[20];

int no=0; int main()

{


FILE *fp1,*fp2;

char fname[10],op[10],ch;

char operand1[8],operand2[8],result[8];

int i=0,j=0;

printf("\n Enter filename of the intermediate code");


scanf("%s",&fname);

fp1=fopen(fname,"r");

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

if(fp1==NULL || fp2==NULL)
```

```
{

printf("\n Error opening the file"); exit(0);
}

while(!feof(fp1))

{
fprintf(fp2,"\n");
fscanf(fp1,"%s",op);
i++;
if(check_label(i))

fprintf(fp2,"\nlabel#%d",i);
if(strcmp(op,"print")==0)
{

fscanf(fp1,"%s",result);

fprintf(fp2,"\n\t OUT %s",result);
}
if(strcmp(op,"goto")==0)
{
fscanf(fp1,"%s %s",operand1,operand2);
fprintf(fp2,"\n\t JMP %s,label#%s",operand1,operand2);
label[no++]=atoi(operand2);
}
if(strcmp(op,"[]=")==0)
{
```

```
fscanf(fp1,"%s %s %s",operand1,operand2,result);
fprintf(fp2,"\n\t STORE %s[%s],%s",operand1,operand2,result);
}
if(strcmp(op,"uminus")==0)
{
fscanf(fp1,"%s %s",operand1,result);
fprintf(fp2,"\n\t LOAD -%s,R1",operand1);
fprintf(fp2,"\n\t STORE R1,%s",result);
}
switch(op[0])
{
case '*':          fscanf(fp1,"%s %s %s",operand1,operand2,result);
                        fprintf(fp2,"\n \tLOAD",operand1);
                        fprintf(fp2,"\n \t LOAD%s,R1",operand2);
                        fprintf(fp2,"\n \t MUL R1,R0");
                        fprintf(fp2,"\n \t STORE R0,%s",result);
                        break;
case '+': fscanf(fp1,"%s %s %s",operand1,operand2,result);
                    fprintf(fp2,"\n \t LOAD %s,R0",operand1);
                    fprintf(fp2,"\n \t LOAD %s,R1",operand2);
                    fprintf(fp2,"\n \t ADD R1,R0");
                    fprintf(fp2,"\n \t STORE R0,%s",result);
                    break;
case '-': fscanf(fp1,"%s %s %s",operand1,operand2,result);
                    fprintf(fp2,"\n\t LOAD %s,R0",operand1);
                    fprintf(fp2,"\n \tLOAD %s,R1",operand2);
                    fprintf(fp2,"\n \t SUB R1,R0");
                    fprintf(fp2,"\n \t STORE R0,%s",result);
                    break;
```

```
case '/': fscanf(fp1,"%s %s s",operand1,operand2,result);
                    fprintf(fp2,"\n \t LOAD %s,R0",operand1);
                    fprintf(fp2,"\n \t LOAD %s,R1",operand2);
                    fprintf(fp2,"\n \t DIV R1,R0");
                    fprintf(fp2,"\n \t STORE R0,%s",result);
                    break;

case '%': fscanf(fp1,"%s %s %s",operand1,operand2,result);
                    fprintf(fp2,"\n \t LOAD %s,R0",operand1);
                    fprintf(fp2,"\n \t LOAD %s,R1",operand2);
                    fprintf(fp2,"\n \t DIV R1,R0");
                    fprintf(fp2,"\n \t STORE R0,%s",result);
                    break;

case '=': fscanf(fp1,"%s %s",operand1,result);
                    fprintf(fp2,"\n\t STORE %s %s",operand1,result);
                    break;

case '>': j++;
                    fscanf(fp1,"%s %s %s",operand1,operand2,result);
                    fprintf(fp2,"\n \t LOAD %s,R0",operand1);
                    fprintf(fp2,"\n\t JGT %s,label#%s",operand2,result);
                    label[no++]=atoi(result);
                    break;

case '<': fscanf(fp1,"%s %s %s",operand1,operand2,result);
                    fprintf(fp2,"\n \t LOAD %s,R0",operand1);
                    fprintf(fp2,"\n\t JLT %s,label#%d",operand2,result);
                    label[no++]=atoi(result);
                    break;
```

```
}
}

fclose(fp2);
fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("Error opening the file\n"); exit(0);
}
do
{
ch=fgetc(fp2); printf("%c",ch);
}while(ch!=EOF); fclose(fp1);
return 0;
}
int check_label(int k)
{
int i; for(i=0;i<no;i++)
{
if(k==label[i]) return 1;
}
return 0;
}
```

input.txt

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

Output:

```
hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$ ./a.out

 Enter filename of the intermediate codeinput.txt


        LOAD t2,R0
        LOAD t2,R1
        DIV R1,R0
        STORE R0,♦U

        LOAD -t2,R1
        STORE R1,t2

        OUT t2

        LOAD t3,R0
        LOAD t4,R1
        ADD R1,R0
        STORE R0,print

♦hadoop@hadoop-virtual-machine:~/Desktop/Compiler design lab$
```

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