COMPILER DESIGN LAB RECORD

Course Code: 19CSE401

CH.EN.U4CSE22025
Name: JEBA RACHEL NESICA

Academic Year: 2022 - 2026

Index

Experiment No: 1

Date: 23/6/25

Implementation of Lexical Analyzer Using Lex Tool

Aim: To build a lexical analyzer that classifies tokens such as keywords, identifiers, numbers, strings, comments, operators and delimiters.

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:

1. analyzer.l

```
%{
#include<stdio.h>
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
```

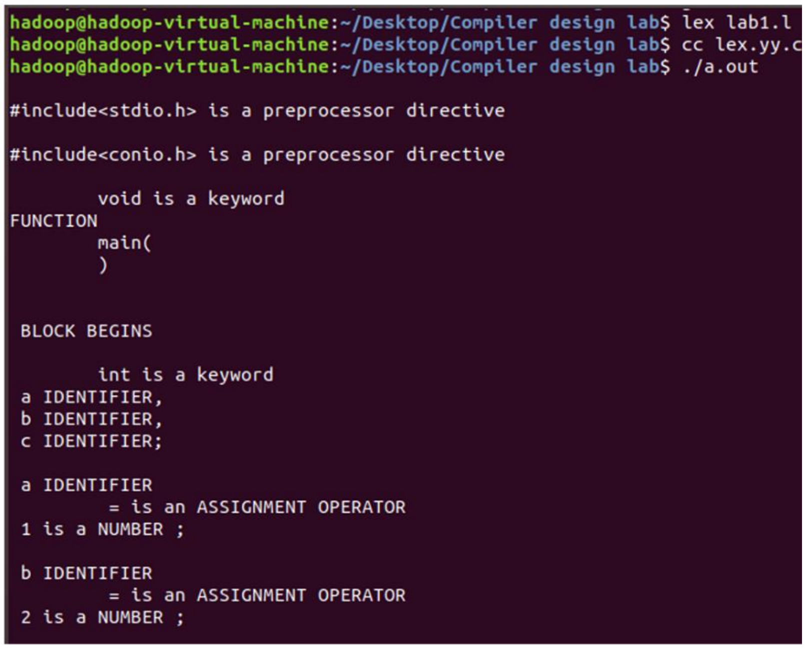
```
#.* {printf("\n%s is a preprocessor directive", yytext);}
int|float|char|double|while|for|struct|typedef|do|if|break|continue|void|switch|return|else|goto
{printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;}{printf("\n\t%s is a COMMENT", yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION \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"); // This program is hardcoded to read "var.c"
   if(!file)
   {
      printf("could not open the file");
      exit(0);
   }
   yyin=file;
   yylex();
   printf("\n");
   return(0);
}
int yywrap()
{
   return(1);
}
```

2. var.c

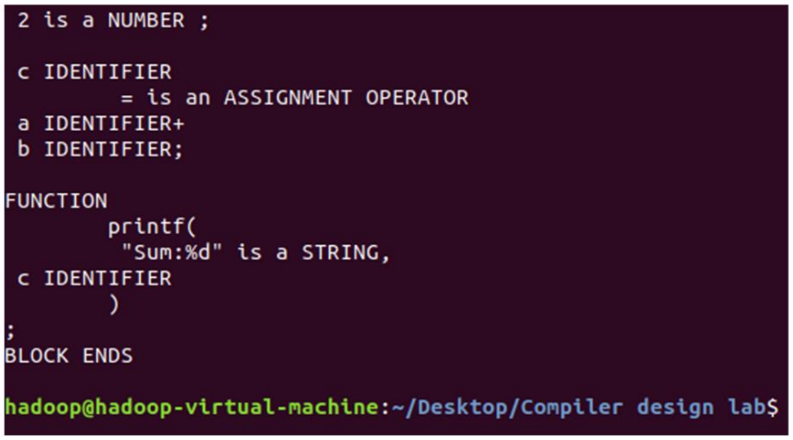
```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a,b,c;
```

```
  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 program executed successfully and produced the expected output.

Experiment No: 2

Date: 7/7/25

Eliminate Left Recursion and Left Factoring (C Program)

Aim: To transform a grammar by removing immediate left recursion and applying left factoring for LL(1) parsing.

Algorithm:

1. Parse a single production of the form A->Aα|β (left recursion) or A->αβ1|αβ2 (left factoring).
2. For left recursion: rewrite as A->βA' and A'->αA'|ε.
3. For left factoring: factor common prefix α into A->αA' and A'->β1|β2.
4. Print the transformed grammar.

Code:

leftrec_simple.C

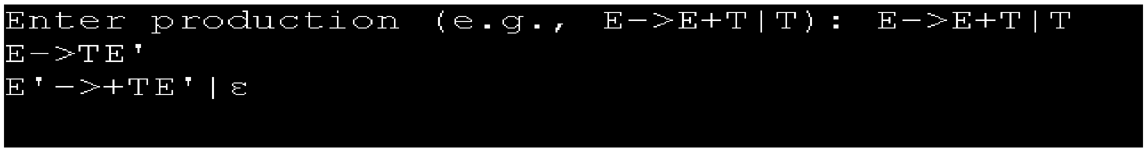
```
#include <stdio.h>
#include <string.h>
int main(void){
   char prod[256], left[128], right[128]; char A;
   printf("Enter production (e.g., E->E+T|T): ");
   if(!fgets(prod, sizeof prod, stdin)) return 0;
   A = prod[0];
   char *arrow = strstr(prod,"->");
   char *bar = strchr(prod,'|');
   if(!arrow||!bar){ puts("Bad input"); return 0; }
   int n = (int)(bar-(arrow+2));
   strncpy(left,  arrow+2, n); left[n]=0;
   strcpy(right, bar+1); right[strcspn(right,"\r\n")]=0;
   if(left[0]!=A && right[0]==A){ char tmp[128]; strcpy(tmp,left); strcpy(left,right);
strcpy(right,tmp); }
   if(left[0]!=A){ puts("No immediate left recursion."); printf("%c->%s|%s\n",A,left,right);
return 0; }
```

```
    printf("%c->%s%c'\n",A,right,A);
    printf("%c'->%s%c'|ε\n",A,left+1,A);
    return 0;
}
```

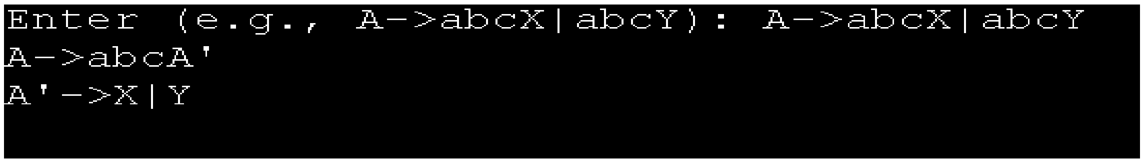
factor.C

```
#include <stdio.h>
#include <string.h>
int main(void){
    char line[512]; printf("Enter (e.g., A->abcX|abcY): ");
    if(!fgets(line,sizeof line,stdin)) return 0;
    char A=line[0]; char *arrow=strstr(line,"->"); char *bar=strchr(line,'|');
    if(!arrow||!bar){ puts("Bad input"); return 0; }
    char p1[256]={0}, p2[256]={0};
    strncpy(p1,arrow+2,bar-(arrow+2)); strcpy(p2,bar+1);
    p2[strcspn(p2,"\r\n")]=0;
    int i=0; while(p1[i]&&p2[i]&&p1[i]==p2[i]) i++;
    if(i==0){ printf("%c->%s|%s\n",A,p1,p2); return 0; }
    printf("%c->%.*s%c'\n",A,i,p1,A);
    printf("%c'->%s|%s\n",A,p1+i,p2+i);
    return 0;
}
```

Output:



```
Enter production (e.g., E->E+T|T): E->E+T|T
E->TE'
E'->+TE'|ε
```



```
Enter (e.g., A->abcX|abcY): A->abcX|abcY
A->abcA'
A'->X|Y
```

Result:

The program executed successfully and produced the expected output.

Experiment No: 3

Date: 14/7/25

Implementation of LL(1) Parsing (C Program)

Aim: To parse input strings using a table-driven LL(1) predictive parser.

Algorithm

1. Initialize stack with $ and start symbol.

2. At each step, if top is terminal, match with lookahead; else consult parse table M[A,a].

3. Push RHS in reverse order; for ε, push nothing.

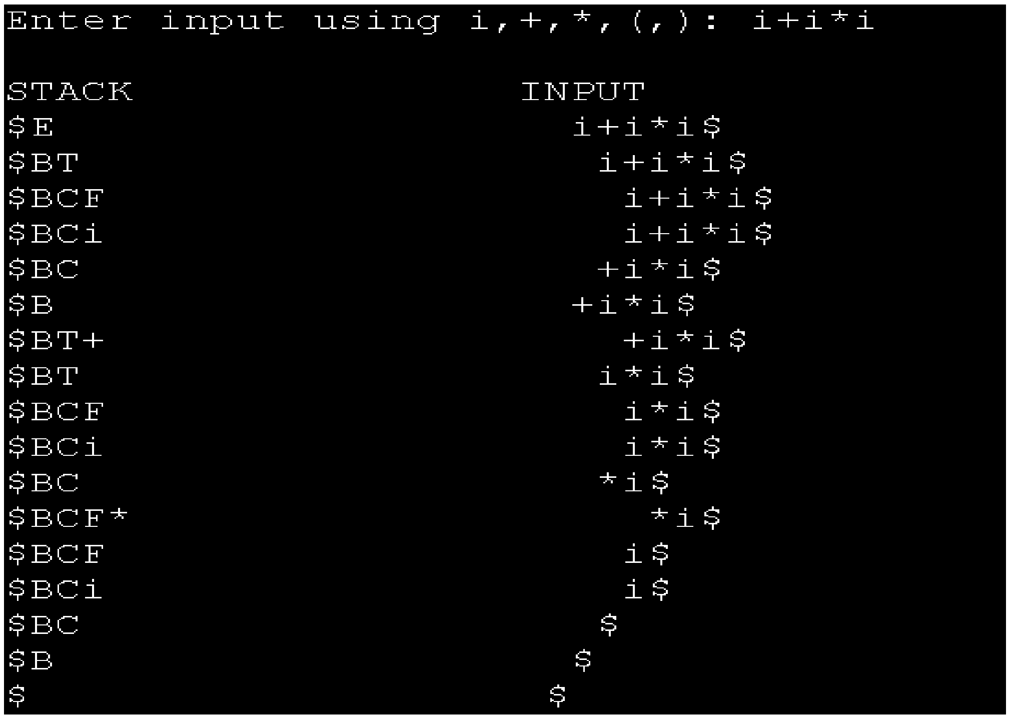
4. Accept when both stack and input have $.

Code

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


static const char* M[5][6] = {
   {"TB","","","TB","",""},
   {"","+TB","","","n","n"},
   {"FC","","","FC","",""},
   {"","n","*FC","","n","n"},
   {"i","","","(E)","",""}
};
int nt_row(char X){ switch(X){case 'E':return 0;case 'B':return 1;case 'T':return 2;case
'C':return 3;case 'F':return 4;} return -1; }
int tm_col(char a){ switch(a){case 'i':return 0;case '+':return 1;case '*':return 2;case '(':return
3;case ')':return 4;case '$':return 5;} return -1; }
int is_term(char x){ return (x=='i'||x=='+'||x=='*'||x=='('||x==')'||x=='$'); }
int main(void){
   char input[256], stack[256]; int top=0, ip=0;
   printf("Enter input using i,+,*,(,): "); scanf("%255s", input); strcat(input,"$");
```

```
    stack[top++]='$'; stack[top++]='E';
    printf("\n%-20s%-20s\n","STACK","INPUT");
    while(top>0){
        for(int k=0;k<top;k++) putchar(stack[k]); printf("%-20s",""); printf("%s\n", input+ip);
        char X=stack[top-1], a=input[ip];
        if(X=='$'&&a=='$'){ puts("\nSUCCESS"); break; }
        if(is_term(X)){ if(X==a){ top--; ip++; } else { puts("ERROR"); return 0; } continue; }
        int r=nt_row(X), c=tm_col(a); if(r<0||c<0){ puts("ERROR"); return 0; }
        const char* rhs = M[r][c]; if(rhs[0]=='\0'){ puts("ERROR"); return 0; }
        top--;
        if(rhs[0]!='n'){ for(int i=(int)strlen(rhs)-1;i>=0;--i) stack[top++]=rhs[i]; }
    }
    return 0;
}
```

Output:



```
Enter input using i,+,*,(,): i+i*i

STACK                INPUT
$E                   i+i*i$
$BT                   i+i*i$
$BCF                   i+i*i$
$BCi                   i+i*i$
$BC                   +i*i$
$B                   +i*i$
$BT+                   +i*i$
$BT                   i*i$
$BCF                   i*i$
$BCi                   i*i$
$BC                   *i$
$BCF*                   *i$
$BCF                   i$
$BCi                   i$
$BC                   $
$B                   $
$                    $
```

Result:

The program executed successfully and produced the expected output.

Experiment No: 4

Date: 25/8/25

Parser Generation using YACC

Aim: To validate arithmetic expressions using a YACC-generated parser with precedence and associativity.

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

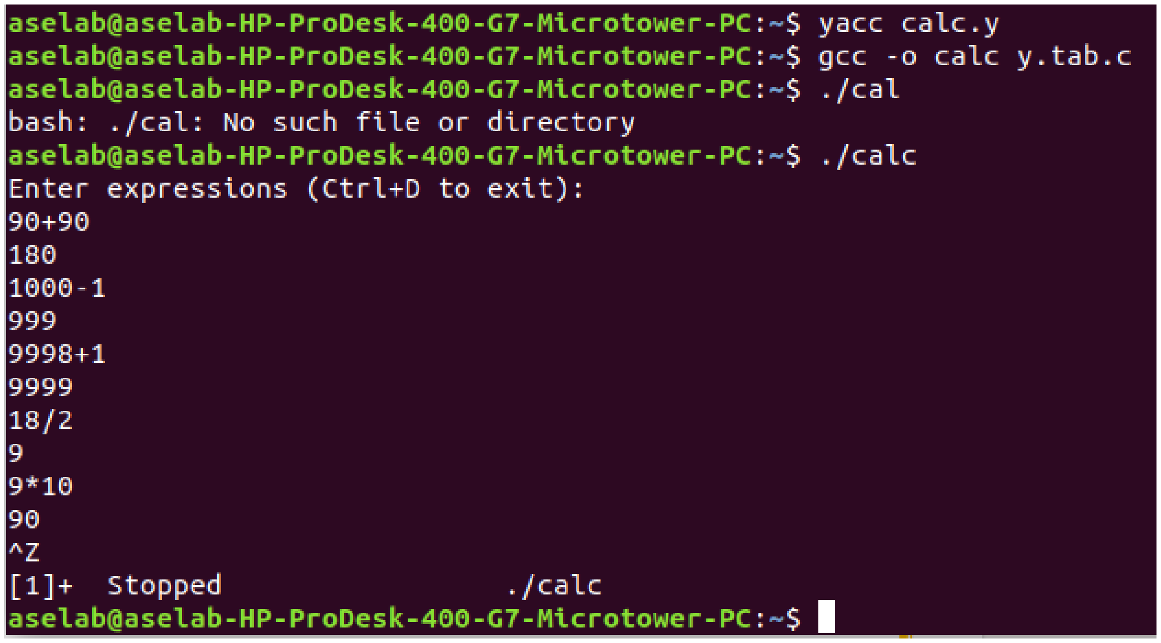
Code:

calc.y

```
%{
#include <stdio.h>
#include <ctype.h>
int yylex(void);
void yyerror(const char *s);
#define YYSTYPE double
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines:
    lines expr '\n'   { printf("Result: %g\n", $2); }
  | lines '\n'
  | /* empty */
  ;

    expr '+' expr     { $$ = $1 + $3; }
  | expr '-' expr     { $$ = $1 - $3; }
  | expr '*' expr     { $$ = $1 * $3; }
  | expr '/' expr     {
                        if ($3 == 0) {
                            yyerror("Division by zero");
                            $$ = 0;
                        } else {
                            $$ = $1 / $3;
                        }
                      }
  | '(' expr ')'      { $$ = $2; }
  | '-' expr %prec UMINUS { $$ = -$2; }
  | NUMBER
  ;
%%

// Simple lexer
int yylex(void) {
    int c;

    // Skip whitespace
    while ((c = getchar()) == ' ' || c == '\t');

    // Handle numbers
    if (c == '.' || isdigit(c)) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }

    return c;
}

// Error handling
```

```
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int yywrap() {
    return 1;
}

int main() {
    printf("Enter expressions (CTRL+D to quit):\n");
    yyparse();
    return 0;
}
```

Output:



```
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ yacc calc.y
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ gcc -o calc y.tab.c
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ ./cal
bash: ./cal: No such file or directory
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ ./calc
Enter expressions (Ctrl+D to exit):
90+90
180
1000-1
999
9998+1
9999
18/2
9
9*10
90
^Z
[1]+  Stopped                 ./calc
aselab@aselab-HP-ProDesk-400-G7-Microtower-PC:~$ 
```

Result:

The program executed successfully and produced the expected output.

Experiment No: 5

Date: 1/9/25

Implementation of Symbol Table

Aim: To implement a basic symbol table supporting insert, lookup, update, delete and display operations.

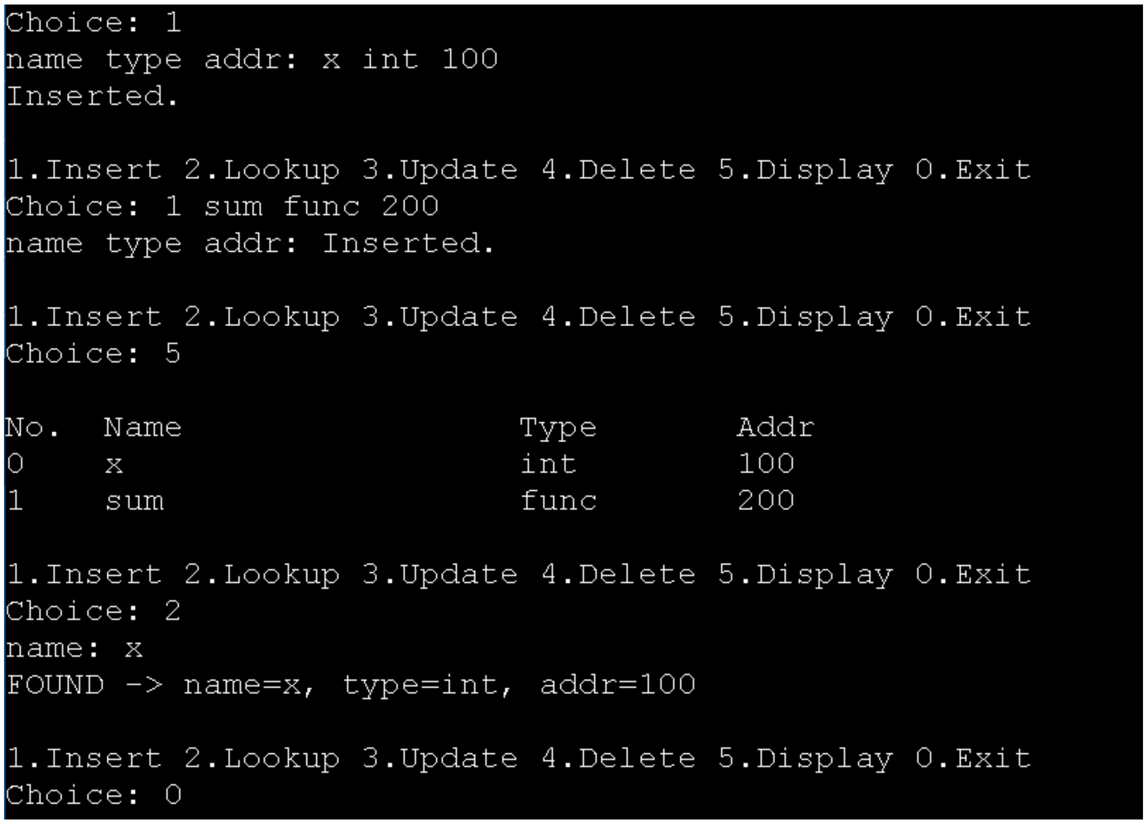
Algorithm

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 <string.h>
#define MAXSYM 100
typedef struct { char name[32]; char type[16]; int addr; } Symbol;
Symbol table[MAXSYM]; int count=0;

int find(const char*name){ for(int i=0;i<count;i++) if(strcmp(table[i].name,name)==0) return
i; return -1; }
void insert(const char*name,const char*type,int addr){
   if(count>=MAXSYM){ puts("Table full!"); return; }
   if(find(name)!=-1){ puts("Duplicate!"); return; }
   strcpy(table[count].name,name); strcpy(table[count].type,type); table[count].addr=addr;
count++; puts("Inserted.");
}
void lookup(const char*name){
    int i=find(name); if(i==-1) puts("Not found.");
    else printf("FOUND -> name=%s, type=%s, addr=%d\n", table[i].name, table[i].type,
table[i].addr);
}
void update(const char*name,const char*type,int addr){
    int i=find(name); if(i==-1){ puts("Not found."); return; }
    strcpy(table[i].type,type); table[i].addr=addr; puts("Updated.");
}
void delete(const char*name){
    int i=find(name); if(i==-1){ puts("Not found."); return; }
    for(int j=i;j<count-1;j++) table[j]=table[j+1]; count--; puts("Deleted.");
}
void display(void){
    if(count==0){ puts("Table empty."); return; }
    printf("\nNo.  Name                 Type       Addr\n");
    for(int i=0;i<count;i++) printf("%-4d %-20s %-10s %-6d\n", i, table[i].name, table[i].type,
table[i].addr);
}
int main(void){
    int ch,addr; char name[32],type[16];
    for(;;){
        printf("\n1.Insert 2.Lookup 3.Update 4.Delete 5.Display 0.Exit\nChoice: ");
        if(scanf("%d",&ch)!=1) break; if(ch==0) break;
        switch(ch){
            case 1: printf("name type addr: "); scanf("%31s %15s %d",name,type,&addr);
insert(name,type,addr); break;
            case 2: printf("name: "); scanf("%31s",name); lookup(name); break;
            case 3: printf("name new_type new_addr: "); scanf("%31s %15s %d",name,type,&addr);
update(name,type,addr); break;
            case 4: printf("name: "); scanf("%31s",name); delete(name); break;
            case 5: display(); break;
            default: puts("Invalid.");
        }
    } return 0;
}
```

Output:



```
Choice: 1
name type addr: x int 100
Inserted.

1.Insert 2.Lookup 3.Update 4.Delete 5.Display 0.Exit
Choice: 1 sum func 200
name type addr: Inserted.

1.Insert 2.Lookup 3.Update 4.Delete 5.Display 0.Exit
Choice: 5

No.  Name                 Type       Addr
0    x                    int        100
1    sum                  func       200

1.Insert 2.Lookup 3.Update 4.Delete 5.Display 0.Exit
Choice: 2
name: x
FOUND -> name=x, type=int, addr=100

1.Insert 2.Lookup 3.Update 4.Delete 5.Display 0.Exit
Choice: 0
```

Result:

The program executed successfully and produced the expected output.

Experiment No: 6

Date: 8/9/25

Implementation of Intermediate Code Generation

Aim: To generate three-address code using temporaries by converting infix expressions to postfix and emitting operations.

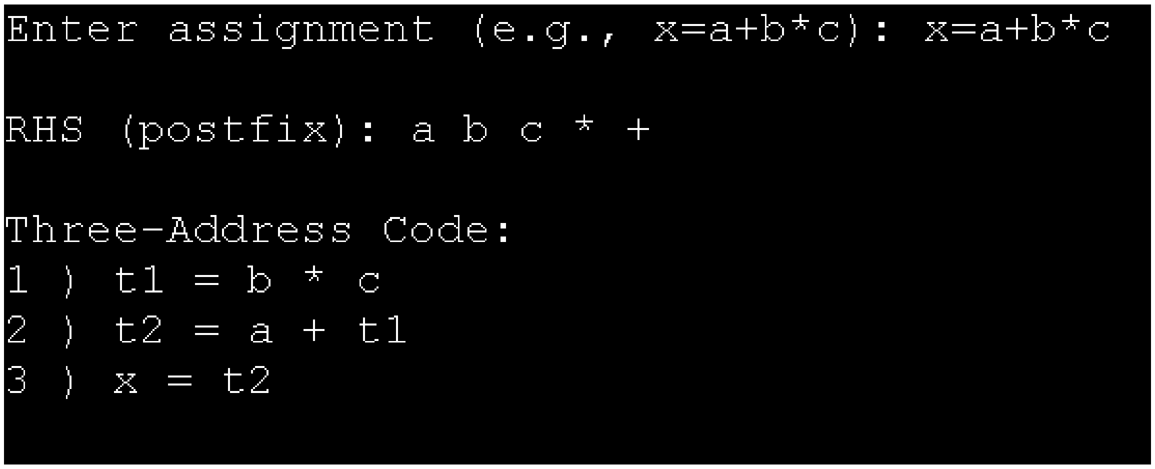
Algorithm:

1. Split input into LHS and RHS of an assignment.
2. Convert RHS to postfix (Shunting-Yard).
3. Scan postfix; for each operator, emit t=arg1 op arg2 and push t.
4. Assign the final temporary to LHS.

Code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 256
int is_op(char c){ return (c=='+'||c=='-'||c=='*'||c=='/'); }
int prec(char c){ return (c=='+'||c=='-')?1:(c=='*'||c=='/')?2:0; }
void infix_to_postfix(const char*in,char*out){
   char stk[MAX]; int top=-1,i=0,k=0;
   while(in[i]){
      if(isspace((unsigned char)in[i])){ i++; continue; }
      if(isalpha((unsigned char)in[i])||in[i]=='_'||isdigit((unsigned char)in[i])){
         while(isalpha((unsigned char)in[i])||isdigit((unsigned char)in[i])||in[i]=='_')
out[k++]=in[i++];
         out[k++]=' '; continue;
      }
      if(in[i]=='('){ stk[++top]=in[i++]; continue; }
      if(in[i]==')'){ while(top>=0&&stk[top]!='('){ out[k++]=stk[top--]; out[k++]=' '; }
if(top>=0) top--; i++; continue; }
      if(is_op(in[i])){
         while(top>=0 && is_op(stk[top]) && prec(stk[top])>=prec(in[i])){ out[k++]=stk[top--
]; out[k++]=' '; }
            stk[++top]=in[i++]; continue;
        } i++;
    } while(top>=0){ if(stk[top]!='('){ out[k++]=stk[top]; out[k++]=' '; } top--; } out[k]=0;
}
void gen_tac(const char*lhs,const char*post){
    char st[MAX][32]; int sp=0; int t=1, q=1; char tok[64]; int i=0;
    printf("\nThree-Address Code:\n");
    while(sscanf(post+i,"%63s",tok)==1){
        i += (int)strcspn(post+i," ")+1;
        if(strlen(tok)==1 && is_op(tok[0])){
            char b[32],a[32],temp[32]; if(sp<2){ puts("ERROR"); return; }
            strcpy(b,st[--sp]); strcpy(a,st[--sp]); snprintf(temp,sizeof temp,"t%d",t++);
            printf("%-2d) %s = %s %c %s\n", q++, temp, a, tok[0], b);
            strcpy(st[sp++], temp);
        }else strcpy(st[sp++], tok);
    }
    if(sp>0) printf("%-2d) %s = %s\n", q++, lhs, st[sp-1]);
}
int main(void){
    char line[MAX]; printf("Enter assignment (e.g., x=a+b*c): ");
    if(!fgets(line,sizeof line,stdin)) return 0;
    char *eq=strchr(line,'='); if(!eq){ puts("Bad input"); return 0; }
    char lhs[64], rhs[MAX]; int i=0,j=0;
    while(line[i] && !isspace((unsigned char)line[i]) && line[i]!='=') lhs[j++]=line[i++];
lhs[j]=0;
    eq++; while(*eq && isspace((unsigned char)*eq)) eq++; strncpy(rhs,eq,sizeof rhs-1);
rhs[sizeof rhs-1]=0;
    for(int k=(int)strlen(rhs)-1;k>=0 && isspace((unsigned char)rhs[k]);--k) rhs[k]=0;
    char post[MAX]={0}; infix_to_postfix(rhs,post);
    printf("\nRHS (postfix): %s\n", post[0]?post:"<empty>"); gen_tac(lhs,post); return 0;
}
```

Output:



```
Enter assignment (e.g., x=a+b*c): x=a+b*c

RHS (postfix): a b c * +

Three-Address Code:
1 ) t1 = b * c
2 ) t2 = a + t1
3 ) x = t2
```

Result

The program executed successfully and produced the expected output.

Experiment No: 7

Date: 15/9/25

Implementation of Code Optimization Techniques

Aim: To apply constant folding, algebraic simplification, common subexpression elimination, and dead code elimination on TAC.

Algorithm:

1. Read TAC statements of the form: t = a op b or x = y.

2. Apply constant folding and algebraic identities.

3. Detect and eliminate repeated computations.

4. Remove dead temporaries that are never used downstream.

Code:

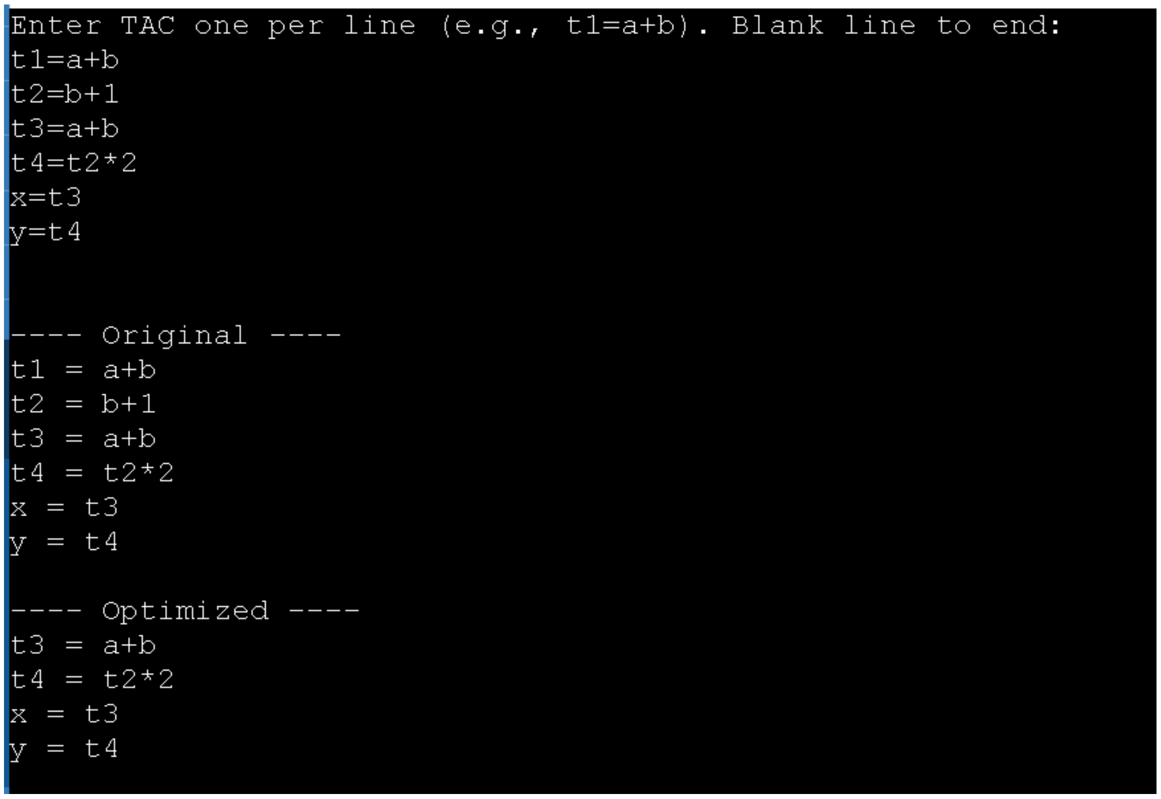
```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#define MAXN 200
#define MAXS 32
typedef struct { char res[MAXS]; char op; char a1[MAXS], a2[MAXS]; int alive; } TAC;
static TAC code[MAXN]; static int n=0;
int is_number(const char*s){ if(!*s) return 0; int i=0; if(s[i]=='+'||s[i]=='-') i++;
if(!isdigit((unsigned char)s[i])) return 0; for(;s[i];++i) if(!isdigit((unsigned char)s[i])) return 0;
return 1; }
long to_long(const char*s){ return strtol(s,NULL,10); }
int is_temp(const char*s){ return s[0]=='t' && isdigit((unsigned char)s[1]); }
void trim(char*s){ int i=0,j=(int)strlen(s)-1; while(i<=j && isspace((unsigned char)s[i])) i++;
while(j>=i && isspace((unsigned char)s[j])) j--; memmove(s,s+i,j-i+1); s[j-i+1]=0; }
int parse_line(const char*line,TAC*t){
  char buf[256]; strcpy(buf,line); char*eq=strchr(buf,'=');
  if(!eq) return 0; *eq=0; trim(buf); trim(eq+1);
  strncpy(t->res,buf,MAXS-1); t->res[MAXS-1]=0;
  char rhs[256]; strcpy(rhs,eq+1); char x1[64]={0}, xop[4]={0}, x2[64]={0};
```

```
    int k=sscanf(rhs," %63s %3s %63s",x1,xop,x2);
    if(k==1){ t->op='='; strncpy(t->a1,x1,MAXS-1); t->a1[MAXS-1]=0; t->a2[0]=0; t-
>alive=1; return 1; }
    else if(k==3 && strlen(xop)==1 && strchr("+-*/",xop[0])){
        t->op=xop[0]; strncpy(t->a1,x1,MAXS-1); t->a1[MAXS-1]=0; strncpy(t->a2,x2,MAXS-1);
t->a2[MAXS-1]=0; t->alive=1; return 1;
    } return 0;
}
void fold_and_simplify(void){
    for(int i=0;i<n;i++){ TAC*t=&code[i]; if(!t->op) continue;
        if(strchr("+-*/",t->op)&&is_number(t->a1)&&is_number(t->a2)){
            long x=to_long(t->a1),y=to_long(t->a2),r=0;
            switch(t->op){ case '+':r=x+y;break; case '-':r=x-y;break; case '*':r=x*y;break; case
'/': if(y!=0) r=x/y; else continue; }
            t->op='='; snprintf(t->a1,sizeof t->a1,"%ld",r); t->a2[0]=0; continue;
        }
        if(t->op=='+'){ if(is_number(t->a1)&&!strcmp(t->a1,"0")){ t->op='='; strcpy(t->a1,t-
>a2); t->a2[0]=0; }
                        else if(is_number(t->a2)&&!strcmp(t->a2,"0")){ t->op='='; t->a2[0]=0; } }
        else if(t->op=='-'){ if(is_number(t->a2)&&!strcmp(t->a2,"0")){ t->op='='; t->a2[0]=0; }
}
        else if(t->op=='*'){ if((is_number(t->a1)&&!strcmp(t->a1,"0"))||(is_number(t-
>a2)&&!strcmp(t->a2,"0"))){ t->op='='; strcpy(t->a1,"0"); t->a2[0]=0; }
                             else if(is_number(t->a1)&&!strcmp(t->a1,"1")){ t->op='='; strcpy(t->a1,t-
>a2); t->a2[0]=0; }
                             else if(is_number(t->a2)&&!strcmp(t->a2,"1")){ t->op='='; t->a2[0]=0; }
                             else if(is_number(t->a2)&&!strcmp(t->a2,"2")){ t->op='+'; strcpy(t->a2,t-
>a1); }
                             else if(is_number(t->a1)&&!strcmp(t->a1,"2")){ t->op='+'; strcpy(t->a1,t-
>a2); } }
        else if(t->op=='/'){ if(is_number(t->a2)&&!strcmp(t->a2,"1")){ t->op='='; t->a2[0]=0; }
}
    }
}
void make_key(const TAC*t,char*out){
    char A[MAXS],B[MAXS]; strcpy(A,t->a1); strcpy(B,t->a2);
    if(t->op=='+'||t->op=='*'){ if(strcmp(A,B)>0){ char tmp[MAXS]; strcpy(tmp,A);
strcpy(A,B); strcpy(B,tmp);} }
    snprintf(out,3*MAXS,"%c %s %s",t->op,A,B);
}
```

```
void common_subexpr_elim(void){
   typedef struct { char key[3*MAXS]; char temp[MAXS]; } Entry;
   Entry seen[MAXN]; int m=0;
   for(int i=0;i<n;i++){ TAC*t=&code[i]; if(!t->op||t->op=='=') continue;
      char key[3*MAXS]; make_key(t,key); int f=-1; for(int j=0;j<m;j++)
if(strcmp(seen[j].key,key)==0){f=j;break;}
      if(f==-1){ strcpy(seen[m].key,key); strcpy(seen[m].temp,t->res); m++; }
      else { t->op='='; strcpy(t->a1,seen[f].temp); t->a2[0]=0; }
   }
}
int is_temp_name(const char*s){ return s[0]=='t' && isdigit((unsigned char)s[1]); }
void dead_code_elim(void){
   typedef struct { char name[MAXS]; int cnt; } Use; Use uses[MAXN]; int u=0;
   for(int i=0;i<n;i++){ TAC*t=&code[i]; if(!t->op) continue;
      char const* list1[] = { t->op=='='?t->a1:t->a1, t->op=='='?"":t->a2, NULL };
      for(int j=0; list1[j]; ++j){ const char* s=list1[j]; if(!*s) continue;
         int k=-1; for(int v=0;v<u;v++) if(strcmp(uses[v].name,s)==0){ k=v; break; }
         if(k==-1){ strcpy(uses[u].name,s); uses[u].cnt=1; u++; } else uses[k].cnt++; }
   }
   for(int i=0;i<n;i++){ TAC*t=&code[i]; if(!t->op) continue;
      if(!is_temp_name(t->res)){ t->alive=1; continue; }
      int used=0; for(int v=0;v<u;v++) if(strcmp(uses[v].name,t->res)==0){ used=1; break; }
      t->alive = used;
   }
}
void print_code(const char*title){
   printf("\n%s\n", title);
   for(int i=0;i<n;i++){ TAC*t=&code[i]; if(!t->op||!t->alive) continue;
      if(t->op=='=') printf("%s = %s\n", t->res, t->a1);
      else printf("%s = %s %c %s\n", t->res, t->a1, t->op, t->a2);
   }
}
int main(void){
   char line[256]; puts("Enter TAC one per line (e.g., t1=a+b). Blank line to end:");
   while(n<MAXN && fgets(line,sizeof line,stdin)){ if(line[0]=='\n'||line[0]=='\r') break;
      TAC t={0}; if(parse_line(line,&t)) code[n++]=t; else puts("! Skipped (bad line)");
   }
   for(int i=0;i<n;i++) code[i].alive=1;
   print_code("---- Original ----"); fold_and_simplify(); common_subexpr_elim();
dead_code_elim(); print_code("---- Optimized ----"); return 0;
```

}

Output:



```
Enter TAC one per line (e.g., t1=a+b). Blank line to end:
t1=a+b
t2=b+1
t3=a+b
t4=t2*2
x=t3
y=t4


---- Original ----
t1 = a+b
t2 = b+1
t3 = a+b
t4 = t2*2
x = t3
y = t4

---- Optimized ----
t3 = a+b
t4 = t2*2
x = t3
y = t4
```

Result:

The program executed successfully and produced the expected output.

Experiment No: 8

Date: 22/9/25

Implementation of Target Code Generation

Aim: To generate simple accumulator-machine target code (LOAD/OP/STORE) from arithmetic assignments.

Algorithm:

1. Split input into LHS and RHS of assignment.

2. Convert RHS to postfix to get evaluation order.

3. For each binary op, emit LOAD a; OP b; STORE Tn; push Tn.

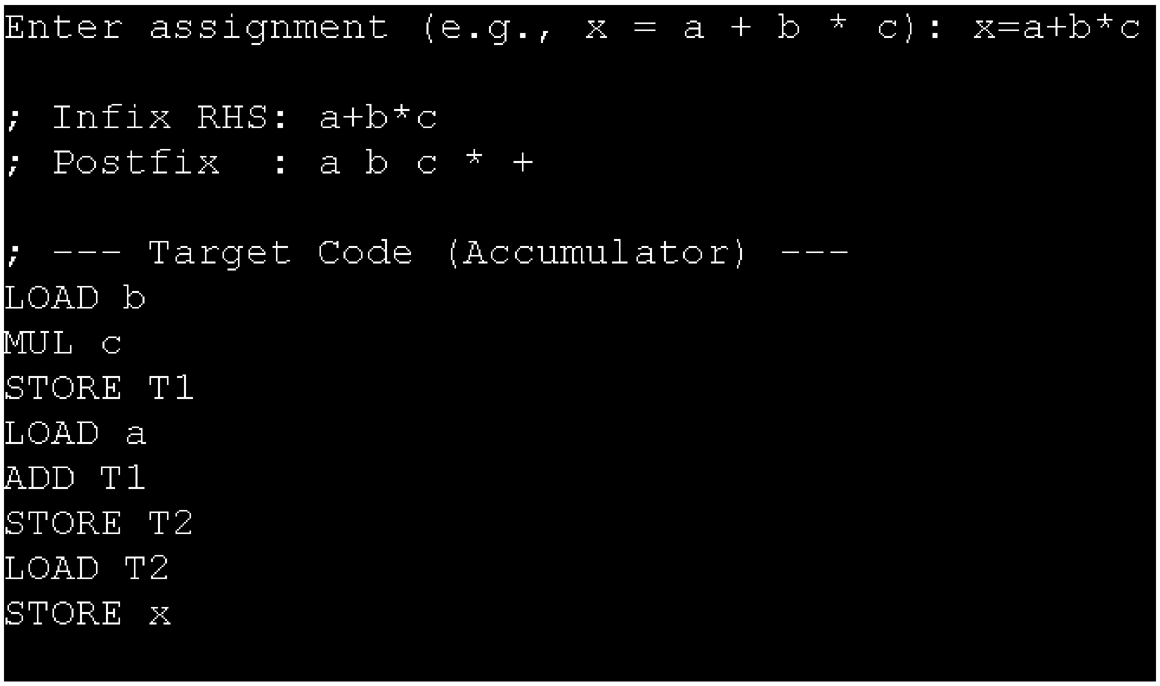
4. Finally store result into LHS.

Code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 256
int is_op(char c){ return (c=='+'||c=='-'||c=='*'||c=='/'); }
int prec(char c){ return (c=='+'||c=='-')?1:(c=='*'||c=='/')?2:0; }
void infix_to_postfix(const char*in,char*out){
   char st[MAX]; int top=-1,i=0,k=0;
   while(in[i]){
      if(isspace((unsigned char)in[i])){ i++; continue; }
      if(isalpha((unsigned char)in[i])||in[i]=='_'||isdigit((unsigned char)in[i])){
         while(isalpha((unsigned char)in[i])||isdigit((unsigned char)in[i])||in[i]=='_')
out[k++]=in[i++];
         out[k++]=' '; continue;
      }
      if(in[i]=='('){ st[++top]=in[i++]; continue; }
      if(in[i]==')'){ while(top>=0&&st[top]!='('){ out[k++]=st[top--]; out[k++]=' '; }
if(top>=0) top--; i++; continue; }
      if(is_op(in[i])){ while(top>=0&&is_op(st[top])&&prec(st[top])>=prec(in[i])){
```

```
out[k++]=st[top--]; out[k++]=' '; } st[++top]=in[i++]; continue; }
        i++;
    } while(top>=0){ if(st[top]!='('){ out[k++]=st[top]; out[k++]=' '; } top--; } out[k]=0;
}
void gen_target(const char*lhs,const char*post){
    char st[MAX][32]; int sp=0; int tid=1; char tok[64]; int i=0;
    printf("\n; --- Target Code (Accumulator) ---\n");
    while(sscanf(post+i,"%63s",tok)==1){
        i += (int)strcspn(post+i," ")+1;
        if(strlen(tok)==1 && is_op(tok[0])){
            if(sp<2){ puts("; ERROR"); return; }
            char b[32],a[32],T[32]; strcpy(b,st[--sp]); strcpy(a,st[--sp]); snprintf(T,sizeof
T,"T%d",tid++);
            printf("LOAD %s\n", a);
            switch(tok[0]){ case '+': printf("ADD %s\n", b); break; case '-': printf("SUB %s\n", b);
break;
                            case '*': printf("MUL %s\n", b); break; case '/': printf("DIV %s\n", b); break; }
            printf("STORE %s\n", T);
            strcpy(st[sp++], T);
        }else strcpy(st[sp++], tok);
    }
    if(sp==1){ printf("LOAD %s\n", st[0]); printf("STORE %s\n", lhs); }
}
int main(void){
    char line[MAX]; printf("Enter assignment (e.g., x = a + b * c): ");
    if(!fgets(line,sizeof line,stdin)) return 0;
    char *eq=strchr(line,'='); if(!eq){ puts("Bad input"); return 0; }
    char lhs[64], rhs[MAX]; int i=0,j=0;
    while(line[i] && isspace((unsigned char)line[i])) i++;
    while(line[i] && line[i] != '=' && j < (int)sizeof(lhs)-1) lhs[j++] = line[i++];
    while(j>0 && isspace((unsigned char)lhs[j-1])) j--; lhs[j]=0;
    eq++; while(*eq && isspace((unsigned char)*eq)) eq++; strncpy(rhs,eq,sizeof rhs-1);
rhs[sizeof rhs-1]=0;
    for(int k=(int)strlen(rhs)-1;k>=0 && isspace((unsigned char)rhs[k]); --k) rhs[k]=0;
    char post[MAX]={0}; infix_to_postfix(rhs,post);
    printf("\n; Infix RHS: %s\n; Postfix  : %s\n", rhs, post[0]?post:"<empty>");
    gen_target(lhs,post); return 0;
}
```

Output:



```
Enter assignment (e.g., x = a + b * c): x=a+b*c

; Infix RHS: a+b*c
; Postfix  : a b c * +

; --- Target Code (Accumulator) ---
LOAD b
MUL c
STORE T1
LOAD a
ADD T1
STORE T2
LOAD T2
STORE x
```

Result:

The program executed successfully and produced the expected output.