

PART A**1. a. Program to count the number of characters, words, spaces and lines in a given input file.**

```
%{
    /*This is definition sec*/
    int cc=0,wc=0,sc=0,lc=0;
}%
    /*This is rules sec*/
%%%%
[ ] {sc++;cc++;}
[ ^ \t\n ]+ {wc++;cc+=yyleng;}
[ \n ] {lc++;cc++;}
[ \t ] {sc+=8;cc++;}
%%%%
/*This is user defined subroutine section*/
int main()
{
    char fname[20];
    printf("Enter a file name:\n");
    scanf("%s",fname);
    yyin=fopen(fname,"r");
    yylex();
    fclose(yyin);
    printf("The no.of characters = %d\n",cc);
    printf("The no.of spaces = %d\n",sc);
    printf("The no.of lines = %d\n",lc);
    printf("The no.of words = %d\n",wc);
}
```

Output:-

```
[student@localhost ~]# vi ex
Note: A file will be opened enter some lines of sentences in it. Come back by typing Esc
:wq
[student@localhost ~]# lex Prg1A.l
[student@localhost ~]# cc lex.yy.c -ll
[student@localhost ~]# ./a.out
```

Enter a file Name: ex

The no.of characters = 40
 The no.of spaces = 5
 The no.of lines = 3
 The no.of words = 7

1b. Program to count the numbers of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file.

```
%{
int com=0;
%}

%%%[^\n]+%%*/ {com++;fprintf(yyout," ");}

int main()
{
    Printf("write a c program\n");
    yyout=fopen("output","w");
    yylex();
    printf("comment=%d\n",com);
    return 0;
}
```

Output:-

```
[student@localhost ~]# lex Prg1B.l
[student@localhost ~]# cc lex.yy.c -ll
[student@localhost ~]# ./a.out
```

Write a c program

```
#include<stdio.h>
int main()
{
// Simple Program
Printf(" Hii ");
}
Ctrl +d
Comment=1
```

2. a. Program to recognize a valid arithmetic expression and to recognize the identifiers and operators present print them separately.

```
%{
    int
    opdcount=0,oprcount=0,i=0,j=0,k=0;
    char OPD[20],OPR[20];
%}

OPERATOR [+\\-\\*\\/]
OPERANDS [a-zA-Z0-9]+

%%%
{OPERATOR} {oprcount++;opr[i++]=yytext[o];opr[i+]=';'}
{OPERANDS} {opdcount++;for(j=0;j<yylen;j++) opd[k++]=yytext[j];
opd[k+1]=';'}
. {}
%%

int main()
{
    yylex();
    IF((OPDCOUNT-OPRCOUNT)!=1)
        printf("Not valid expression\n");
    else
    {
        printf("Valid\n");
        printf("the operands = %s and count = %d\n",opd,opdcount);
        printf("The operators = %s and count = %d\n",opr,oprcount);
    }
    return 1;
}
```

```
[student@localhost ~]# lex Prg2A.l
[student@localhost ~]# cc lex.yy.c -ll
[student@localhost ~]# ./a.out
```

```
a+b*(e*d)/f*(h-g)
Valid
The operands = a b e d f h g and count = 7
The operators = + * * / * - and count=6
```

2. b Write a LEX Program to scan reserved word & Identifiers of C Language.

```
%{
    int count=0;
%}

SPLCH [!@#$%^&] DIGIT [0-9]
LETTER [a-zA-Z]
USCORE "_"

%%%
(({LETTER}|{USCORE})+({LETTER}{DIGIT}|{USCORE})*) {count++;}. {}
(({LETTER}|{DIGIT}|{USCORE}|{SPLCH})+({LETTER}|{DIGIT}|{USCORE}|{SPLCH})*) {}

%%

int main()
{
    yyin=fopen("ex","r");
    system("cat ex");
    yylex();
    fclose(yyin);
    printf("no of identifiers = %d",count);
    return 1;
}
```

Output:-

```
[student@localhost ~]# lex 2B.1
[student@localhost ~]# cc lex.yy.c -ll
[student@localhost ~]# ./a.out
1243_4664
filename a
_sdsds
no data
no. of identifiers = 5
```

3. a Write a YACC Program to evaluate an arithmetic expression involving operators +, -, *, and /.

```
%{
#include<stdio.h>
#include<ctype.h>
#define YYSTYPE double
%}

%token num
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines:
|lines exp '\n' {PRINTF("%G\n",$2); return 1;}
|lines '\n'
;
exp:exp+'exp {$$=$1+$3;}
|exp-'exp {$$=$1-$3;}
|exp'*'exp {$$=$1*$3;}
|exp'/'exp {if($3==0)
{
    printf("Divide by zero error\n"); exit(0);
}
$$=$1/$3;
}
|'('exp')' {$$=$2;}
|'-exp {$$=-$2;}
|num;
%%
int yylex()
{
    int c;
    while((c=getchar())==' ');
    if((c=='.')||isdigit(c)))
    {
        ungetc(c,stdin);
        scanf("%lf",&yyval)
        ; return num;
    }
    return c;
}
int main()
{
    yyparse();
    return 0;
}
```

```
}
```

```
int yyerror()
```

```
{
```

```
    printf("Error\n")
```

```
    ; return 0;
```

```
}
```

Output:-

```
[student@localhost ~]# yacc 3.y
```

```
[student@localhost ~]# cc y.tab.c
```

```
[student@localhost ~]# ./a.out
```

```
4+4
```

```
8
```

```
[student@localhost ~]# ./a.out
```

```
12+3*(23-5)
```

```
66
```

```
[student@localhost ~]# ./a.out
```

```
-55+(5*5)
```

```
-30
```

4. Write a YACC Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.

```
%{  
    #include<stdio.h>  
    #include<ctype.h>  
%}  
  
%token letter digit  
%%  
id: id letter '\n' {printf("Valid\n");}  
| id letter other '\n' {printf("Valid\n");}  
|  
;  
other: other letter  
| other digit  
| letter  
| digit  
;  
%%  
int main()  
{  
    yyparse();  
    return 0;  
}  
int yyerror()  
{  
    printf("Error\n");  
}  
int yylex()  
{  
    int c;  
    while((c=getchar())==' ');  
    if(isalpha(c)) return letter;  
    if(isdigit(c)) return digit;  
    else return c;  
}
```

Output:-

```
[student@localhost ~]# yacc 4.y
```

```
[student@localhost ~]# cc y.tab.c
[student@localhost ~]# ./a.out
A2772
Valid
67676
Error
```

5. a Write a YACC Program to recognize the grammar ($a^n b$, $n \geq 10$).

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

%token ta tb
%%

v:|
v ta ta ta ta ta ta ta ta p '\n' {printf("Valid\n");}
;

p:ta p
|tb
;

%%

int yylex()
{
    int c;
    while((c=getchar())==' ');
    if(c=='a') return ta;
    if(c=='b') return tb;
    return c;
}

int main()
{
    yyparse();
    return 0;
}

int yyerror()
{
    printf("Error\n");
    return 0;
}
```

Output:-

```
[student@localhost ~]# yacc 5a.y
[student@localhost ~]# cc y.tab.c
```

```
[student@localhost ~]# ./a.out
aaaaaaaaaaab
Valid
aaaaaaaaaaaabbbbbbb
Error
```

5. b YACC Program to recognize strings ‘aaab’, ‘abbb’, ‘ab’ and ‘a’ using the grammar ($a^n b^n$, $n \geq 0$).

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

%token ta tb
%%

S: T '\n' {printf("Valid\n"); exit(0);}
;

T:
|ta T tb
;

int yylex()
{
    int c;
    while((c=getchar())==' ');
    if(c=='a')
        return ta;
    if(c=='b')
        return tb;
    return c;
}

int main()
{
    yyparse();
    return 0;
}

int yyerror()
{
    printf("Error\n");
```

```

        return 0;
    }

```

Output:-

```

[student@localhost ~]# yacc 5B.y
[student@localhost ~]# cc y.tab.c
[student@localhost ~]# ./a.out
aaab
In valid
[student@localhost ~]# ./a.out
aaaaabbbbb
Valid

```

PART B

6. Design, develop and implement program to construct Predictive / LL(1) Parsing Table for the grammar rules:

```

A ->aBa ,
B ->bB |ε . Use this table to parse the sentence: abba$
/*GRAMMER RULES ---- A ->aBa , B ->bB | @*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char prod [3][10]={"A->aBa","B->bB","B->@"};
char first[3][10]={"a","b","@"};
char follow[3][10]={"$","a","a"};
char table[3][4][10];
char input[10];
int top=-1;
char stack[25];
char curp[20];
void push(char item)
{
    stack[++top]=item;
}
void pop()
{
    top=top-1;
}
void display()
{
    int i;
    for(i=top;i>=0;i--)
        printf("%c",stack[i]);
}
int numr(char c)
{

```

```

switch(c)
{
    case'A':return 1;
    case'B':return 2;
    case'a':return 1;
    case'b':return 2;
    case'@':return 3;
}
return 1;
}

int main()
{
    char c;
    int i,j,k,n;
    for(i=0;i<3;i++){
        for(j=0;j<4;j++){
            strcpy(table[i][j],"EMPTY");
        }
    }
    printf("\nGrammar\n");
    for(i=0;i<3;i++)
        printf("%s\n",prod[i]);
    printf("\nfirst=%{%,%,%s},first[0],first[1],first[2]);\n");
    printf("\nfollow=%{%,%s}\n",follow[0],follow[1]);
    printf("\nPredictive parsing table for the given grammar :\n");
    strcpy(table[0][0],"");
    strcpy(table[0][1],"a");
    strcpy(table[0][2],"b");
    strcpy(table[0][3],"$");
    strcpy(table[1][0],"A");
    strcpy(table[2][0],"B");
    for(i=0;i<3;i++)
    {
        if(first[i][0]!='@')
            strcpy(table[numr(prod[i][0])][numr(first[i][0])],prod[i]);
        else
            strcpy(table[numr(prod[i][0])][numr(follow[i][0])],prod[i]);
    }
    printf("\n-----\n");
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
        {
            printf("%-30s",table[i][j]);
            if(j==3) printf("\n-----\n");
        }
    }
    printf("Enter the input string terminated with $ to parse:-");
    scanf("%s",input);
    for(i=0;input[i]!='\0';i++)
    {

```

```
if((input[i]!='a')&&(input[i]!='b')&&(input[i]!='$'))
{
    printf("Invalid String");
    exit(0);
}

if(input[i-1]!='$')
{
    printf("\n\nInput String Entered Without End Marker $");
    exit(0);
}

push('$');
push('A');
i=0;
printf("\n\n");
printf("Stack\t Input\tAction");
printf("\n-----\n");
while(input[i]!='$'&&stack[top]!='$')
{
    display();
    printf("\t\t%s\t", (input+i));
    if(stack[top]==input[i])
    {
        printf("\tMatched %c\n", input[i]);
        pop();
        i++;
    }
    else
    {
        if(stack[top]>=65&&stack[top]<92)
        {
            strcpy(curp,table[numr(stack[top])][numr(input[i])]);
            if(!(strcmp(curp,"e")))
            {
                printf("\nInvalid String - Rejected\n");
                exit(0);
            }
            else
            {
                printf("\tApply production %s\n", curp);
                if(curp[3]=='@')
                pop();
                else
                {
                    pop();
                    n=strlen(curp);
                    for(j=n-1;j>=3;j--)
                    push(curp[j]);
                }
            }
        }
    }
}
```

```

        }
    }
}
}

display();
printf("\t\t%s\t",(input+i));
printf("-----\n");
if(stack[top]=='$'&&input[i]=='$')
{
    printf("\nValid String - Accepted\n");
}
else
{
    printf("Invalid String - Rejected\n");
}
}
}

```

```

test5@localhost:~]
[test5@localhost ~]$ vi lab3.c
[test5@localhost ~]$ cc lab3.c
[test5@localhost ~]$ ./a.out

Grammar
A->aBa
B->bB
B->@

first={a,b,@}
follow={$,a}

Predictive parsing table for the given grammar :

-----
          a           b           $
-----
A          A->aBa      EMPTY      EMPTY
-----
B          B->@       B->bB     EMPTY
-----
Enter the input string terminated with $ to parse:-abba$


Stack   Input  Action
-----
A$      abba$   Apply production A->aBa
aBa$    abba$   Matched a
Ba$     bba$   Apply production B->bB
bBa$    bba$   Matched b
Ba$     ba$    Apply production B->bB
bBa$    ba$    Matched b
Ba$     a$     Apply production B->@
a$      a$     Matched a
$       $      -
-----
Valid String - Accepted

```

7. Design, develop and implement program to demonstrate Shift Reduce Parsing technique for the grammar rules:

$$E \rightarrow E + T | T$$

$$T \rightarrow T * F | F,$$

$F \rightarrow (E) | id$ and parse the sentence: $id + id * id$.

```
#include<stdio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
{
    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("\nEnter input string :");
    gets(a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
    puts("stack \t input \t action");
    for(k=0,i=0; j<c; k++,i++,j++)
    {
        if(a[j]=='i' && a[j+1]=='d')
        {
            stk[i]=a[j];
            stk[i+1]=a[j+1];
            stk[i+2]='\0';
            a[j]=' ';
            a[j+1]=' ';
            printf("\n$%s\t%s$\t%s",stk,a,act);
            check();
        }
        else
        {
            stk[i]=a[j];
            stk[i+1]='\0';
            a[j]=' ';
        }
    }
}
```

```
        printf("\n$%s\t%s$\t%ssymbols",stk,a,act);check();
    }
}
void check()
{
    strcpy(ac,"REDUCE TO E");
    for(z=0; z<c; z++)
        if(stk[z]=='i' && stk[z+1]=='d')
    {
        stk[z]='E';
        stk[z+1]='\0';
        printf("\n$%s\t%s$\t%os",stk,a,ac);
        j++;
    }
    for(z=0; z<c; z++)
        if(stk[z]=='E' && stk[z+1]==',' && stk[z+2]=='E')
    {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+2]='\0';
        printf("\n$%s\t%s$\t%os",stk,a,ac);
        i=i-2;
    }
    for(z=0; z<c; z++)
        if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
    {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+2]='\0';
        printf("\n$%s\t%s$\t%os",stk,a,ac);
        i=i-2;
    }
    for(z=0; z<c; z++)
        if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
    {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+2]='\0';
        printf("\n$%s\t%s$\t%os",stk,a,ac);
        i=i-2;
    }
}
```

```

[test5@localhost ~]$ vi lab4.c
[test5@localhost ~]$ cc lab4.c
/tmp/ccajtIrp.o: In function `main':
lab4.c:(.text+0x31): warning: the `gets' function is dangerous
[test5@localhost ~]$ ./a.out
GRAMMAR is E->E+E
E->E*E
E->(E)
E->id

Enter input string :
id+id*id
stack      input      action

$ id      +id*ids      SHIFT->id
$ E      +id*ids      REDUCE TO E
$ E+
id*ids      SHIFT->symbols
$ E+id      *ids      SHIFT->id
$ E+E      *ids      REDUCE TO E
$ E      *ids      REDUCE TO E
$ E*      ids      SHIFT->symbols
$ E*id      $      SHIFT->id
$ E*E      $      REDUCE TO E
[test5@localhost ~]$ 

```

8. Design, develop and implement the syntax-directed definition of “if E then S1” and “if E then S1 else S2”

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int parsecondition(char[ ],int,char*,int);
void gen(char[ ],char[ ],char[ ],int);
int main()
{
int counter=0,stlen=0,elseflag=0;
char stmt[60];
char strB[54];
char strS1[50];
char strS2[45];
printf("Format of 'if statement\n example..\n");
printf("if(a<b) then(s=a);\n");
printf("if(a<b)then(s=a) else (s=b);\n\n");
printf("Enter the statement\n");
scanf("%[^n]",stmt);
stlen=strlen(stmt);
counter=counter+2;
counter=parsecondition(stmt,counter,strB,stlen);
if(stmt[counter]==')')
counter++;
counter=counter+3;
counter=parsecondition(stmt,counter,strS1,stlen);
if(stmt[counter+1]==';')

```

```
{  
printf("\n parsing the input statement");  
gen(strB,strS1,strS2,elseflag);  
return 0;  
}  
if(stmt[counter]=='')  
counter++;  
counter=counter+3;  
counter=parsecondition(stmt,counter,strS2,stlen);  
counter=counter+2;  
if(counter==stlen)  
{  
elseflag=1;  
printf("\n parsing the input statement");  
gen(strB,strS1,strS2,elseflag);  
return 0;  
}  
return 0;  
}  
int parsecondition(char input[ ],int cntr,char* dest,int totallen)  
{  
int index=0,pos=0;  
while(input[cntr]!='(' && cntr<=totallen)  
cntr++;  
if(cntr>=totallen)  
return 0;  
index=cntr;  
while(input[cntr]!=')')  
cntr++;  
if(cntr>=totallen)  
return 0;  
while(index<=cntr)  
dest[pos++]=input[index++];  
dest[pos]='\0';  
return cntr;  
}  
void gen(char B[ ],char S1[ ],char S2[ ],int elsepart)  
{  
int Bt=101,Bf=102,Sn=103;  
printf("\n\t if %s goto%d",B,Bt);  
printf("\n\tgoto %d",Bf);  
printf("\n %d:",Bt);  
printf("%s",S1);  
if(!elsepart)  
printf("\n%od\n",Bf);  
else  
{  
printf("\n\t goto %d",Sn);  
printf("\n %d:%s",Bf,S2);  
printf("\n%od\n",Sn);
```

}

}

Output:

Format of 'if' statement

example..

```
if(a<b) then(s=a);
if(a<b)then(s=a) else (s=b);
```

Enter the statement

```
if(a<b) then (x=a);
```

parsing the input statement

```
if (a<b) goto101
```

```
goto 102
```

```
101:(x=a)
```

```
102
```

9. Write a yacc program that accepts a regular expression as input and produce its parse tree as output.

```
%/*declaration part*/
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
#define MAX 100 /*to store productions*/
int getREindex ( const char* );
signed char productions[MAX][MAX];
int count = 0 , i , j;
char temp[200] , temp2[200];
%
%token ALPHABET
%left '|'
%left '!'
%nonassoc '*' '+'
%/*rules section*/
S : re '\n' {
printf ( "This is the rightmost derivation--\n" );
for ( i = count - 1 ; i >= 0 ; --i ) {
    if ( i == count - 1 ) {
        printf ( "\nre => " );
        strcpy ( temp , productions[i] );
        printf ( "%s" , productions[i] );
    }
    else {
        printf ( "\n=> " );
    }
}
}
```

```

j = getREindex ( temp );
temp[j] = '\0';
sprintf ( temp2 , "%s%s%s" , temp , productions[i] , (temp + j + 2) );
printf ( "%s" , temp2 );
strcpy ( temp , temp2 );
}
}
printf ( "\n" );
exit ( 0 );
}
re : ALPHABET {
temp[0] = yylval; temp[1] = '\0';
strcpy ( productions[count++], temp );/*copy the input to the production array*/
/*only conditions defined here will be valid, this is the structure*/
| 're' /*adds the (expression) to the production array*/
{ strcpy ( productions[count++], "(re)" ); }
| re '*'
{ strcpy ( productions[count++], "re*" ); }
| re '+' /*adds expression+ type to the production array*/
{ strcpy ( productions[count++], "re+" ); }
| re '|' re /*adds the expression|expression to the production array*/
{ strcpy ( productions[count++], "re | re" ); }
| re '.' re /*adds the expression.expression to the production array*/
{ strcpy ( productions[count++], "re . re" ); }
;
%%
int main ( int argc , char **argv )
{
/*
Parse and output the rightmost derivation,
from which we can get the parse tree
*/
yyparse();/*calls the parser*/
return 0;
}

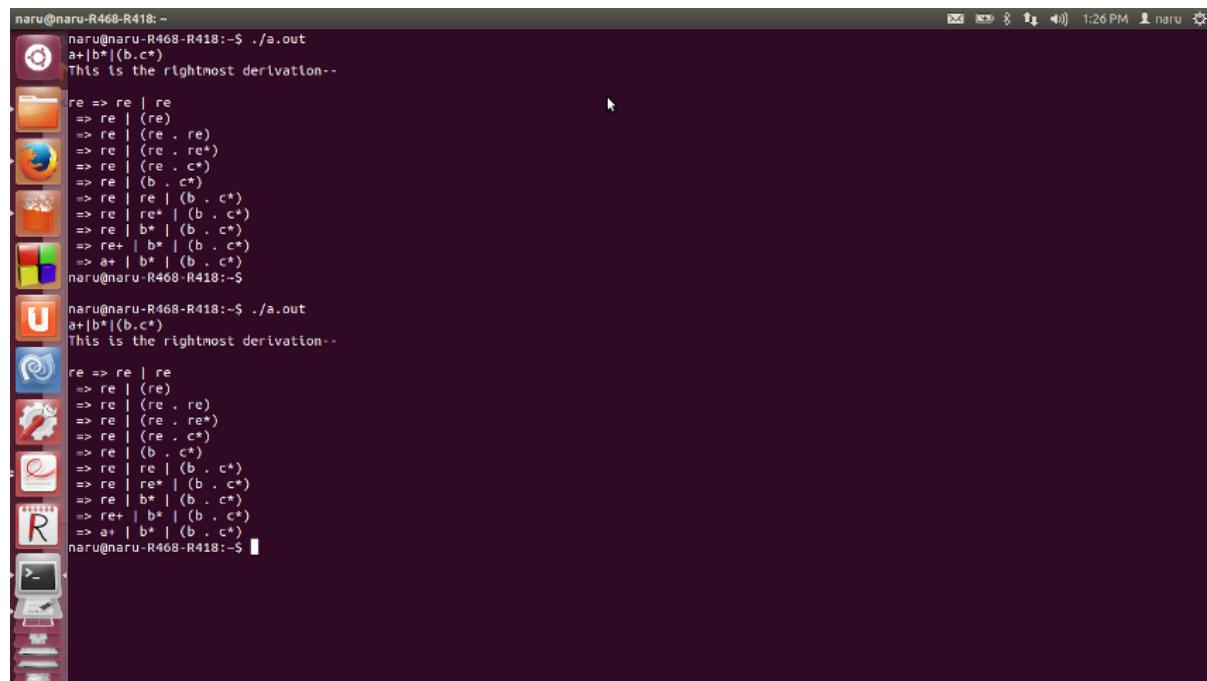
yylex() /*calls lex and takes each character as input and feeds ALPHABET to check for the
structure*/
{
    signed char ch = getchar();
    yylval = ch;
    if ( isalpha ( ch ) )
        return ALPHABET;
    return ch;
}

yyerror() /*Function to alert user of invalid regular expressions*/
{
    fprintf(stderr , "Invalid Regular Expression!!\n");
    exit ( 1 );
}

```

```
}
```

```
int getREindex ( const char *str )
{
    int i = strlen ( str ) - 1;
    for ( ; i >= 0 ; --i ) {
        if ( str[i] == 'e' && str[i-1] == 'r' )
            return i-1;
    }
}
```



```
naru@naru-R468-R418:~  
naru@naru-R468-R418:~$ ./a.out  
a+b*(b,c*)  
This is the rightmost derivation--  
re => re | re  
=> re | (re)  
=> re | (re . re)  
=> re | (re . re*)  
=> re | (re . c*)  
=> re | (b . c*)  
=> re | re | (b . c*)  
=> re | re* | (b . c*)  
=> re | b* | (b . c*)  
=> re+ | b* | (b . c*)  
=> a+ | b* | (b . c*)  
naru@naru-R468-R418:~$  
  
naru@naru-R468-R418:~$ ./a.out  
a+r*(b,c*)  
This is the rightmost derivation--  
re => re | re  
=> re | (re)  
=> re | (re . re)  
=> re | (re . re*)  
=> re | (re . c*)  
=> re | (b . c*)  
=> re | re | (b . c*)  
=> re | re* | (b . c*)  
=> re | b* | (b . c*)  
=> re+ | b* | (b . c*)  
=> a+ | b* | (b . c*)  
naru@naru-R468-R418:~$
```

10. Design, develop and implement a program to generate the machine code using Triples for the statement $A = -B * (C + D)$ whose intermediate code in three-address form:

$$\begin{aligned}T1 &= -B \\T2 &= C + D \\T3 &= T1 + T2 \\A &= T3\end{aligned}$$

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

char op[2],arg1[5],arg2[5],result[5];
int main()
{
    FILE *fp1,*fp2;
    fp1=fopen("input.txt","r");
    fp2=fopen("output.txt","w");
    while(!feof(fp1))
    {
        fscanf(fp1,"%s%s%s%s",result,arg1,op,arg2);
        if(strcmp(op,"+")==0)
        {
            fprintf(fp2,"\nMOV R0,%s",arg1);
            fprintf(fp2,"\nADD R0,%s",arg2);
            fprintf(fp2,"\nMOV %s,R0",result);
        }
        if(strcmp(op,"*")==0)
        {

```

```
fprintf(fp2,"nMOV R0,%s",arg1);
fprintf(fp2,"nMUL R0,%s",arg2);
fprintf(fp2,"nMOV %s,R0",result);
}

if(strcmp(op,"-")==0)
{
    fprintf(fp2,"nMOV R0,%s",arg1);
    fprintf(fp2,"nSUB R0,%s",arg2);
    fprintf(fp2,"nMOV %s,R0",result);
}

if(strcmp(op,"/")==0)
{
    fprintf(fp2,"nMOV R0,%s",arg1);
    fprintf(fp2,"nDIV R0,%s",arg2);
    fprintf(fp2,"nMOV %s,R0",result);
}

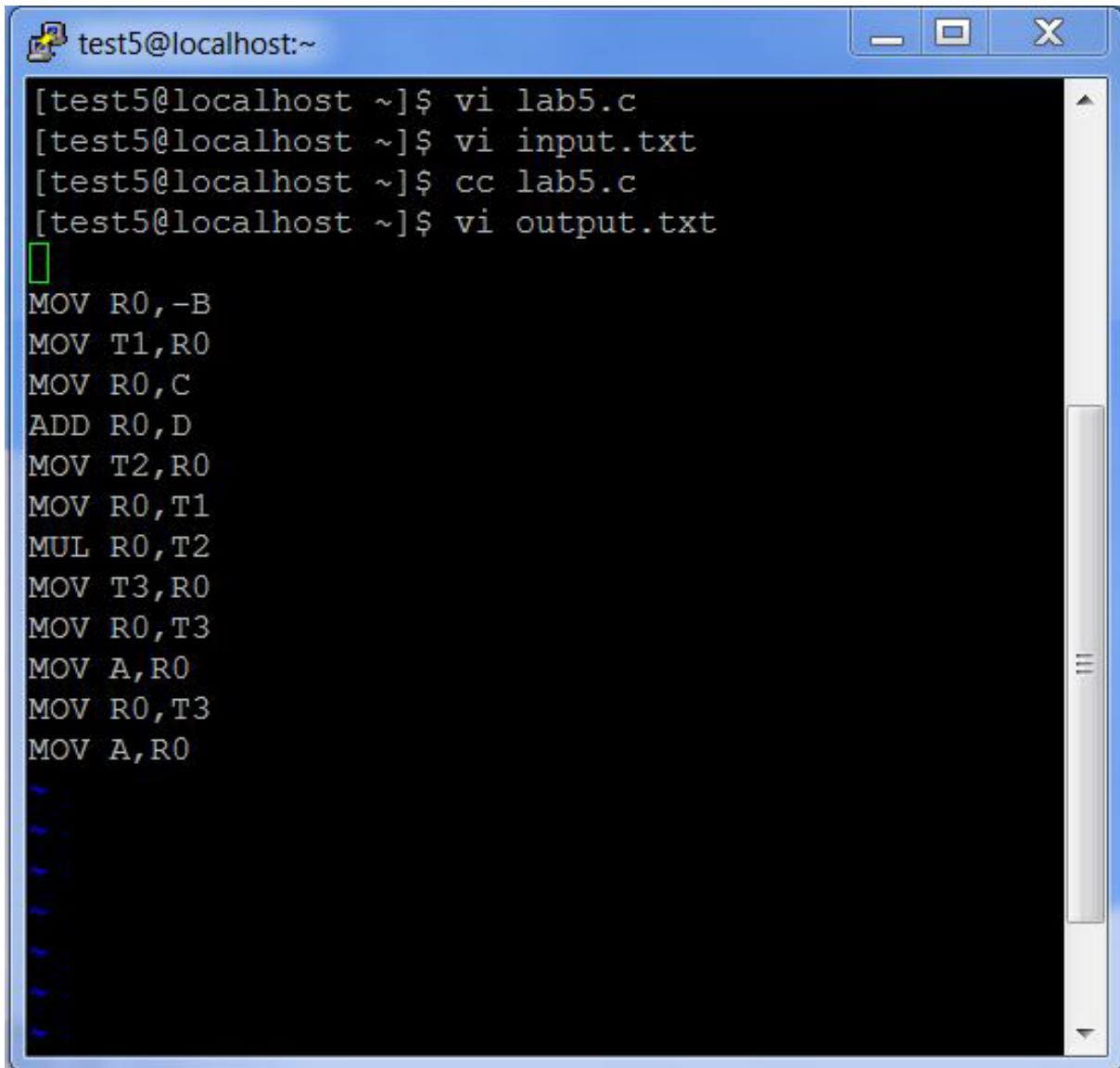
if(strcmp(op,"")==0)
{
    fprintf(fp2,"nMOV R0,%s",arg1);
    fprintf(fp2,"nMOV %s,R0",result);
}

fclose(fp1);
fclose(fp2);
}
```

OUTPUT:

```
gedit prg10.c
cat>input.txt
cc prg10.c
./a.out
cat output.txt
```

```
T1 -B = ?
T2 C + D
T3 T1 * T2
A T3 = ?
```



The screenshot shows a terminal window titled "test5@localhost:~". The window contains the following text:

```
[test5@localhost ~]$ vi lab5.c
[test5@localhost ~]$ vi input.txt
[test5@localhost ~]$ cc lab5.c
[test5@localhost ~]$ vi output.txt
```

Below this, there is a large block of assembly code:

```
MOV R0,-B
MOV T1,R0
MOV R0,C
ADD R0,D
MOV T2,R0
MOV R0,T1
MUL R0,T2
MOV T3,R0
MOV R0,T3
MOV A,R0
MOV R0,T3
MOV A,R0
```