1. Write a C program to implement Recursive Descent parser for the following grammar

S → (L)/a

L → SL’

L’ →, SL’/∈

#include<stdio.h>  
#include<conio.h>  
int c=0;  
char p[20];  
void s();  
void l();  
void lprime();  
void l()  
{  
s();  
lprime();  
}  
void lprime()  
{  
if(p[c]==',')  
{  
c++;  
s();  
lprime();  
}  
}  
void s()  
{  
if(p[c]=='a')  
c++;  
else if(p[c]=='(')  
{  
c++;  
l();  
if(p[c]==')')  
c++;  
else  
c--;  
}  
else  
printf("\nInvalid Expression");  
}  
void main()  
{  
clrscr();  
printf("\nImplementation of RECURSIVE DESCENT PARSER\n");  
printf("\nEnter the Expression:\n");  
scanf("%s",p);  
s();  
if(p[c]=='$')  
printf("\nThe String is accepted");  
else  
printf("\nThe string is rejected");  
getch();  
}

**Recursive Descent Parsing Output Result**  
Implementation of RECURSIVE DESCENT PARSER  
Enter the Expression:  
a  
The string is rejected  
Implementation of RECURSIVE DESCENT PARSER  
Enter the Expression:  
a$  
The String is accepted  
Recursive Descent Parsing OUTPUT  
Implementation of RECURSIVE DESCENT PARSER  
Enter the Expression:  
(a)$  
The String is accepted  
  
Implementation of RECURSIVE DESCENT PARSER  
Enter the Expression:  
b\*c  
Invalid Expression  
The string is rejected

1. Implement C program to demonstrate shift reduce parsing technique the grammar:

E → E + E/E\*E/ E/E /a/b

**ALGORITHM:**

1. Get the input expression and store it in the input buffer.
2. Read the data from the input buffer one at the time.
3. Using stack and push & pop operation shift and reduce symbols with respect to production rules available.
4. Continue the process till symbol shift and production rule reduce reaches the start symbol.
5. Display the Stack Implementation table with corresponding Stack actions with input symbols.

**PROGRAM:**

#include

#include

#include

#include

char ip\_sym[15],stack[15];

int ip\_ptr=0,st\_ptr=0,len,i;

char temp[2],temp2[2];

char act[15];

void check();

void main()

{

clrscr();

printf("\n\t\t SHIFT REDUCE PARSER\n");

printf("\n GRAMMER\n");

printf("\n E->E+E\n E->E/E");

printf("\n E->E\*E\n E->a/b");

printf("\n enter the input symbol:\t");

gets(ip\_sym);

printf("\n\t stack implementation table");

printf("\n stack \t\t input symbol\t\t action");

printf("\n\_\_\_\_\_\_\_\_\t\t\_\_\_\_\_\_\_\_\_\_\_\_\t\t\_\_\_\_\_\_\_\_\_\_\_\_\n");

printf("\n $\t\t%s$\t\t\t--",ip\_sym);

strcpy(act,"shift");

temp[0]=ip\_sym[ip\_ptr];

temp[1]='\0';

strcat(act,temp);

len=strlen(ip\_sym);

for(i=0;i<=len-1;i++)

{

stack[st\_ptr]=ip\_sym[ip\_ptr];

stack[st\_ptr+1]='\0';

ip\_sym[ip\_ptr]=' ';

ip\_ptr++;

printf("\n $%s\t\t%s$\t\t\t%s",stack,ip\_sym,act);

strcpy(act,"shift");

temp[0]=ip\_sym[ip\_ptr];

temp[1]='\0';

strcat(act,temp);

check();

st\_ptr++;

}

st\_ptr++;

check();

}

void check()

{

int flag=0;

temp2[0]=stack[st\_ptr];

temp2[1]='\0';

if((!strcmpi(temp2,"a"))||(!strcmpi(temp2,"b")))

{

stack[st\_ptr]='E';

if(!strcmpi(temp2,"a"))

printf("\n $%s\t\t%s$\t\t\tE->a",stack,ip\_sym);

else

printf("\n $%s\t\t%s$\t\t\tE->b",stack,ip\_sym);

flag=1;

}

if((!strcmpi(temp2,"+"))||(strcmpi(temp2,"\*"))||(!strcmpi(temp2,"/")))

{

flag=1;

}

if((!strcmpi(stack,"E+E"))||(!strcmpi(stack,"E\E"))||(!strcmpi(stack,"E\*E")))

{

strcpy(stack,"E");

st\_ptr=0;

if(!strcmpi(stack,"E+E"))

printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip\_sym);

else

if(!strcmpi(stack,"E\E"))

printf("\n $%s\t\t%s$\t\t\tE->E\E",stack,ip\_sym);

else

if(!strcmpi(stack,"E\*E"))

printf("\n $%s\t\t%s$\t\t\tE->E\*E",stack,ip\_sym);

else

printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip\_sym);

flag=1;

}

if(!strcmpi(stack,"E")&&ip\_ptr==len)

{

printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip\_sym);

getch();

exit(0);

}

if(flag==0)

{

printf("\n%s\t\t\t%s\t\t reject",stack,ip\_sym);

exit(0);

}

return;

}

**OUTPUT:**

                             SHIFT REDUCE PARSER

          GRAMMER

          E->E+E

          E->E/E

          E->E\*E

          E->a/b

          Enter the input symbol: a+b

Stack Implementation Table

         Stack              Input Symbol                  Action

         -------           -----------------               ---------

         $                 a+b$                             --

          $a                  +b$                              shift a

          $E                  +b$                              E->a

         $E+                  b$                              shift +

          $E+b                  $                              shift b

          $E+E                  $                              E->b

         $E                   $                              E->E+E

  $E                    $                              ACCEPT

1. Write a C program to implement Predictive parser for the following grammar

E → TE’

E’→ +TE’/∈

T → FT’

T’→ \*FT’/∈

F → (E)/a

#include<stdio.h>

#include<conio.h>

int isterm(char x)

{

if((x=='a')||(x=='+')||(x=='\*')||(x=='(')||(x==')')||(x=='$'))

return 1;

else

return 0;

}

void main()

{

char table[5][6][4]={{"TA","","","TA","",""},

{"","+TA","","","e","e"},

{"FB","","","FB","",},

{"","e","\*FB","","e","e"},

{"a","","","(E)","",""}};

char stack[20],input[20];

int len,l,i,j,k,nt,t,flag;

clrscr();

printf("Enter the input string to be parsed : ");

scanf("%s",input);

l=strlen(input);

input[l]='$';

input[l+1]='\0';

stack[0]='$';

stack[1]='E';

stack[2]='\0';

i=1;j=0;

printf("\n\n");

printf("------------------------------------------\n");

printf("STACK\tINPUT\tOUTPUT\n");

printf("------------------------------------------\n");

printf("%s\t%s\n",stack,input);

while(!(stack[i]=='$'&&input[0]=='$'))

{

if((stack[i]=='$')&&(input[0]!='$'))

{

printf("Not a sentence\n");

goto a;

}

else if(isterm(stack[i])&&(stack[i]!=input[0]))

{

printf("Not a sentence\n");

goto a;

}

else if(isterm(stack[i])&&(stack[i]==input[0]))

{

stack[i]='\0';

i--;

l=strlen(input);

for(j=0;j<l-1;j++)

{

input[j]=input[j+1];

}

input[l-1]='\0';

flag=0;

}

else

{

switch(stack[i])

{

case 'E': nt=0;break;

case 'A': nt=1;break;

case 'T': nt=2;break;

case 'B': nt=3;break;

case 'F': nt=4;break;

}

switch(input[0])

{

case 'a': t=0;break;

case '+': t=1;break;

case '\*': t=2;break;

case '(': t=3;break;

case ')': t=4;break;

case '$': t=5;break;

}

stack[i]='\0';

i--;

l=strlen(table[nt][t]);

if(l==0)

{

printf("Not a sentence\n");

goto a;

}

else if(strcmp(table[nt][t],"e")==0)

{

}

else

{

flag=1;

for(k=l-1;k>=0;k--)

{

stack[++i]=table[nt][t][k];

}

}

}

if(flag==0)

printf("%s\t%s\n",stack,input);

else

printf("%s\t%s\t%s\n",stack,input,table[nt][t]);

}

printf("Sentence\n");

a:

getch();

}

1. Write a C program to perform Stack Implementation of Operator Precedence Parser for the grammar

E->E+E | E-E | E\*E | E/E | E^E | (E) | a

#include<stdio.h>

#include<conio.h>

int f(char x)

{

int p;

switch(x)

{

case '+': p=2;break;

case '-': p=2;break;

case '\*': p=4;break;

case '/': p=4;break;

case '^': p=4;break;

case '(': p=0;break;

case ')': p=6;break;

case 'a': p=6;break;

case '$': p=0;break;

}

return p;

}

int g(char x)

{

int q;

switch(x)

{

case '+': q=1;break;

case '-': q=1;break;

case '\*': q=3;break;

case '/': q=3;break;

case '^': q=5;break;

case '(': q=5;break;

case ')': q=0;break;

case 'a': q=5;break;

case '$': q=0;break;

}

return q;

}

void main()

{

char stack[20],input[20];

int i,j,l,x,y;

char pop;

clrscr();

printf("Enter the input string to be parsed : ");

scanf("%s",input);

l=strlen(input);

input[l]='$';

input[l+1]='\0';

stack[0]='$';

stack[1]='\0';

i=0;j=0;

printf("\n\n");

printf("------------------------------------------\n");

printf("STACK\tINPUT\tACTION\n");

printf("------------------------------------------\n");

printf("%s\t%s\tINITIAL CONFIGURATION\n",stack,input);

while(!(stack[i]=='$'&&input[0]=='$'))

{

if((stack[i]=='a')&&(input[0]=='a'))

{

printf("Not a sentence\n");

goto a;

}

else if(stack[i]=='a'&&input[0]=='(')

{

printf("Not a sentence\n");

goto a;

}

else if(stack[i]==')'&&input[0]=='a')

{

printf("Not a Sentence\n");

goto a;

}

else if(stack[i]==')'&&input[0]=='(')

{

printf("Not a Sentence\n");

goto a;

}

else if(stack[i]=='('&&input[0]=='$')

{

printf("Not a Sentence\n");

goto a;

}

else if(stack[i]=='$'&&input[0]==')')

{

printf("Not a Sentence\n");

goto a;

}

else

{

if(f(stack[i])<g(input[0])||f(stack[i])==g(input[0]))

{

i++;

stack[i]=input[0];

stack[i+1]='\0';

l=strlen(input);

for(j=0;j<l-1;j++)

{

input[j]=input[j+1];

}

input[l-1]='\0';

printf("%s\t%s\tSHIFT\n",stack,input);

}

else

{

do

{

pop=stack[i];

i--;

}while(!(f(stack[i])<g(pop)));

stack[i+1]='\0';

printf("%s\t%s\tREDUCE\n",stack,input);

}

}

}

printf("ACCEPT\n");

a:

getch();

}

1. Implement a simple intermediate code generator in C program, which produces three address code statements for a given input expression.

#include<stdio.h>

#include<conio.h>

FILE \*fp;

void gettoken();

void chkopr();

int n,num,num1,n1,n2,n3,n4,nm;

struct token

{

char tkn[20];

};

struct token t[20],d[20],m[20],a[20],s[20];

struct temp

{

char tmp[10];

};

struct temp tp[10];

struct t1

{

char \*tmp;

};

struct t1 alpha[10];

void main()

{

num=0;

gettoken();

chkopr();

getch();

}

void gettoken()

{

int i,j;

char c;

clrscr();

fp=fopen("inp.txt","r");

i=0;while(!feof(fp))

{

c=getc(fp);

if(c==',')

break;

t[i].tkn[0]=c;

i++;

}

n=i+1;

printf("\nThe tokens are \n");

for(i=0;i<n;i++)

{

printf("%s",t[i].tkn);

}

}

void chkopr()

{

int i,j=0,p,pntr,flag=0;

num1=0;

printf("\n\n");

for(k=0,j=0;k<n;k++,j++)

{

if(t[k].tkn[0]=='/'

{

j--;

d[j].tkn[0]='D';

k++;

pntr=j;

}

else

{

d[j].tkn[0]=t[k].tkn[0];

}

}

for(p=0;p<3;p++)

{

tp[num].tmp[p]=t[pntr].tkn[0];

pntr++;

}

num++;

nn++;

for(k=0;k<n1;k++)

{

printf("%s",d[k].tkn);

}

num1++;

for(k=0,j=0;k<n1;k++,j++)

{

if(d[k].tkn[0]=='\*')

{

j--;

m[j].tkn[0]='m';

alpha[nn].tmp="M";

k++;

if(flag==0)

{

pntr=j;

flag++;

}

}

else

{

m[j].tkn[0]=d[k].tkn[0];

}

}

n2=j;

nn++;

for(p=0;p<3;p++)

{

tp[num].tmp[p]=d[pntr].tkn[0];

pntr++;

}

num++;

printf("\n");

for(k=0;k<n1;k++)

{

printf("%s",m[k].tkn);

}

num1++;

for(k=0,j=0;k<n2;k++,j++)

{

if(m[k].tkn[0]=='+')

{

j--;

a[j].tkn[0]='A';

alpha[nn].tmp="A";

k++;

pntr=j;

}

else

{

a[j].tkn[0]=m[k].tkn[0];

}

}

n3=j;

for(p=0;p<3;p++)

{

tp[num].tmp[p]=m[pntr].tkn[0];

pntr++;

}

nn++;

num++;

printf("\n");

for(k=0;k<n3;k++)

{

printf("%s",a[k].tkn);

}

num1++;

for(k=0,j=0;k<n3;k++,j++)

{

if(a[k].tkn[0]=='-')

{

j--;

a[j].tkn[0]='s';

alpha[nn].tmp='S';

k++;

pntr=j;

}

else

{

s[j].tkn[0]=a[k].tkn[0];

}

}

n4=j;

for(p=0;p<3;p++)

{

tp[num].tmp[p]=a[pntr].tkn[0];

pntr++;

}

num++;

nn++;

printf("\n");

for(k=0;k<n4;k++)

{

printf("%s",s[k].tkn);

}

num1++;

printf("\n");

for(i=0;i<num1;i++)

{

printf("\n\t%s=%s",alpha[i].tmp,tp[i].tmp);

}

}

1. Implement the back end of the compiler using C program.

#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

int n,i,j;

char a[50][50];

clrscr();

printf("enter the no: intermediate code:");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("enter the 3 address code:%d:",i+1);

for(j=0;j<6;j++)

{

scanf("%c",&a[i][j]);

}

}

printf("the generated code is:");

for(i=0;i<n;i++)

{

printf("\n mov %c,R%d",a[i][3],i);

if(a[i][4]=='-')

{

printf("\n sub %c,R%d",a[i][5],i);

}

if(a[i][4]=='+')

{

printf("\n add %c,R%d",a[i][5],i);

}

if(a[i][4]=='\*')

{

printf("\n mul %c,R%d",a[i][5],i);

}

if(a[i][4]=='/')

{

printf("\n div %c,R%d",a[i][5],i);

}

printf("\n mov R%d,%c",i,a[i][1]);

printf("\n");

}

getch();

}

1. Implement simple code optimization techniques using Cprogram.

**Program:**

#include<stdio.h>

#include<conio.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;

clrscr();

printf("Enter the Number of Values:");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("left: ");

op[i].l=getche();

printf("\tright: ");

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",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';

strcpy(pr[i].r,'\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);

}

}

getch();

}

**Sample Input & Output**:

Enter the Number of Values: 5

Left: aright: 9

Left: bright: c+d

Left: eright: c+d

Left: fright: b+e

Left: rright: f

Intermediate Code

a=9

b=c+d

e=c+d

f=b+e

r=:f

After Dead Code Elimination

b=c+d

e=c+d

f=b+e

r=:f

Eliminate Common Expression

b=c+d

b=c+d

f=b+b

r=:f

Optimized Code

b=c+d

f=b+b

r=:f

**Result:**

Thus the code optimization techniques were implemented