**EXP 6(IMPLEMENTING LL PARSER)**

```
from collections import deque

table = {
    "E" : {
        "id" : ["T", "E1"],
        "(" : ["T", "E1"]
    },
    "E1" : {
        "+" : ["+", "T", "E1"],
        ")" : [""],
        "$" : [""]
    },
    "T" : {
        "id" : ["F", "T1"],
        "(" : ["F", "T1"]
    },
    "T1" : {
        "+" : [""],
        "*" : ["*", "F", "T1"],
        ")" : [""],
        "$" : [""]
    },
    "F" : {
        "id" : ["id"],
        "(" : ["(", "E", ")"]
    }
}

stack = ["$", "E"]
terminals = {"id", "+", "*", "(", ")", "$"}
inputstr = deque(input("Enter the input statement:\n").split())
inputstr.append("$")

while True:
    try:
        if stack[-1] == "$" and inputstr[0] == "$":
            print("Success!!")
            break
```

```
        elif stack[-1] in terminals:
            if stack[-1] == inputstr[0]:
                stack.pop()
                inputstr.popleft()
            else:
                print(stack[-1], inputstr[0])
                print("Failure!!")
                break
        else:
            x = stack.pop()
            stack.extend(filter(None, table[x][inputstr[0]][::-1]))
    except:
        print("Failure!!")
        break
```

[***Steps for giving input -***

1) LEAVE A SPACE BETWEEN EACH **CHARACTER** OF THE INPUT
]

[**Tips for code** -
The variable table denotes the **transition table** which is being implemented as a **nested dictionary** where the,

***keys of outer most dictionary*** represent the ***variables (***a.k.a ***the rows of our transition table)*** and the

***keys of the inner dictionary*** denote the ***terminals (*** a.k.a ***the columns of our transition table)*** and the

***Values of the inner most dictionary*** represent the *list* of values which we fill in the ***cells of transition table***
]

**EXP 7(IMPLEMENTING LR PARSER)**

**Version 1 - Directly defining the tables**

ACTION = {

```
    0 : {'a' : 3, 'b' : 4},
    1 : {'$' : '*'},
    2 : {'a' : 6, 'b' : 7},
    3 : {'a' : 3, 'b' : 4},
    4 : {'a' : -3, 'b' : -3},
    5 : {'$' : -1},
    6 : {'a' : 6, 'b' : 7},
    7 : {'$' : -3},
    8 : {'a' : -2, 'b' : -2},
    9 : {'$' : -2}

}


GOTO = {
    0 : {'S' : 1, 'A' : 2},
    2 : {'A' : 5},
    3 : {'A' : 8},
    6 : {'A' : 9}

}

GRAMMAR = [
    None,
    ('S' , ['A', 'A']),
    ('A' , ['a', 'A']),
    ('A' , ['b'])
    ]

sentence = input("Enter the statement: ").strip().split()
stack = ['$', 0]
i = 0
while True:
    if i >= len(sentence) or sentence[i] not in ACTION[stack[-1]]:
        print("Failed!!")
        break
    if ACTION[stack[-1]][sentence[i]] == '*':
        print("Success!!")
        break
    elif ACTION[stack[-1]][sentence[i]] < 0:
```

```
        A, B = GRAMMAR[-ACTION[stack[-1]][sentence[i]]]
        for _ in range(2*len(B)):
            stack.pop()
        stack.append(A)
        stack.append(GOTO[stack[-2]][A])
    else:
        stack.append(sentence[i])
        stack.append(ACTION[stack[-2]][sentence[i]])
        i += 1
```

[***Steps for giving input -***

1) FOR GIVING INPUT STRING, LEAVE A SPACE BETWEEN EACH CHARACTER. ALSO INCLUDE $ AT THE END
]

**Version 2 - Dynamically defining the tables**

```
nG=int(input("Enter the number of productions:"))
G=[]
print("Enter the productions:")
for i in range(nG):
    G.append(input())

T=['$']
NT=[]
for i in G:
    j=i.split("->")
    if j[0] not in NT and j[0].isupper():
            NT.append(j[0])
    y=j[1]
    for x in y:
            if x not in T and not(x.isupper()):
                    T.append(x)

n=int(input("Enter the number of states:"))
Action={}
Goto={}
print("Enter acc for accepted.")
for i in range(n):
```

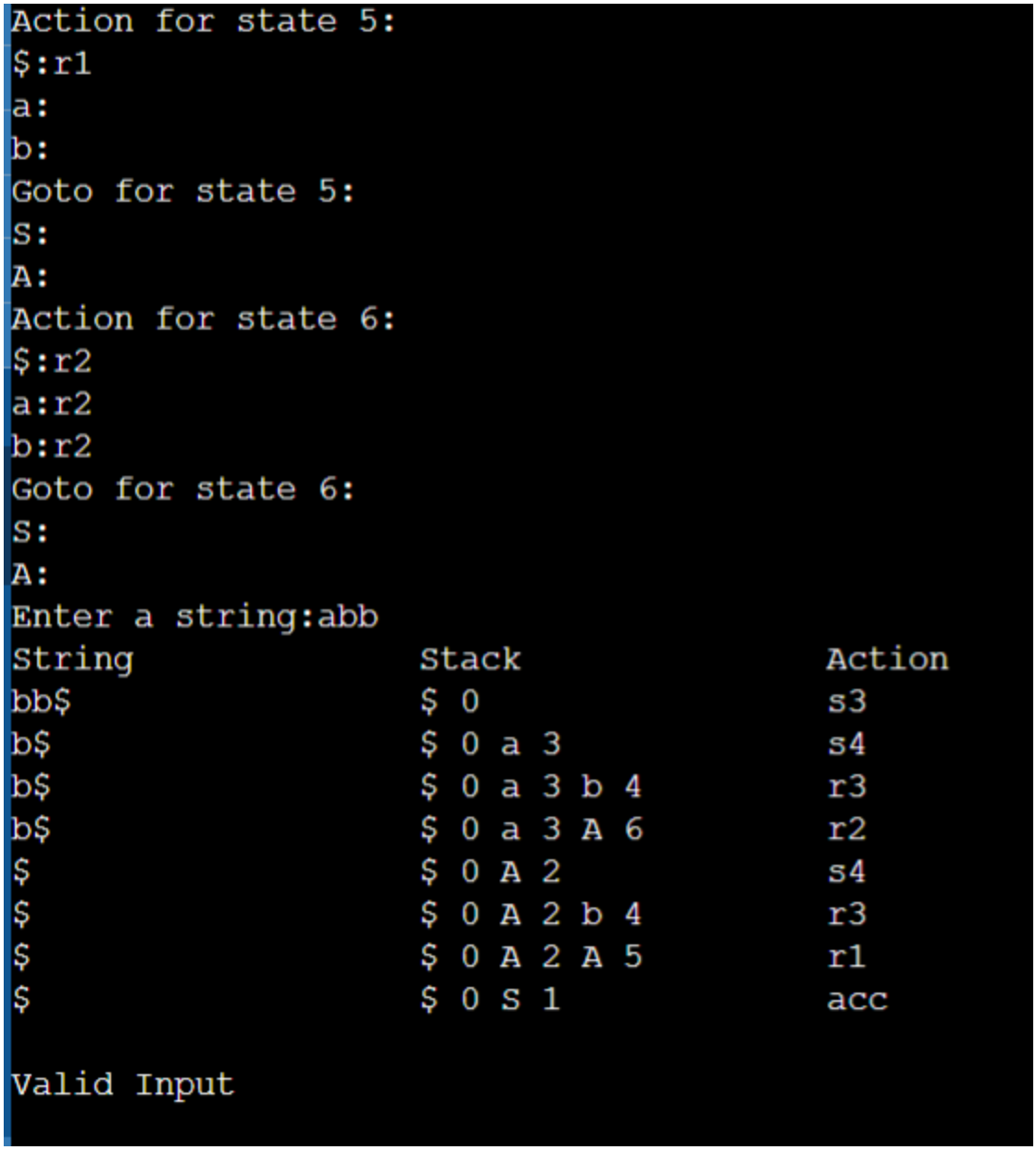
```
    print("Action for state "+str(i)+":")
    for j in T:
        Action[(str(i),j)]=input(j+":")
    print("Goto for state "+str(i)+":")
    for j in NT:
        Goto[(str(i),j)]=input(j+":")

inp=input("Enter a string:")
inp=inp+"$"
print(("{:20}"*3).format("String","Stack","Action"))
i=0
Stack=["$","0"]
flag=0
while True:
    word=inp[i]
    State=Stack[-1]
    StackP=" ".join(Stack)
    key=(State,word)
    if key not in Action or Action[key]=="":
        print("\nInvalid input")
        break
    ac=Action[key]
    if ac=="acc":
        flag=1
    elif ac[0]=='s':
        Stack.extend([word,ac[1:]])
        i+=1
    elif ac[0]=='r':
        r=int(ac[1:])-1
        g=G[r]
        X=g.split("->")
        Y=X[1]
        Stack=Stack[:-(2*len(Y))]
        key=(Stack[-1],X[0])
        if key not in Goto or Goto[key]=="":
            print("\nInvalid input")
            break
        Go=Goto[key]
        Stack.extend([X[0],Go])
    print(("{:<20}"*3).format(inp[i:],StackP,ac))
```

```
    if flag==1:
            print("\nValid Input")
            break
```

Sample input output format

```
Enter the number of productions:3
Enter the productions:
S->AA
A->aA
A->b
Enter the number of states:7
Enter acc for accepted.
Action for state 0:
$:
a:s3
b:s4
Goto for state 0:
S:1
A:2
Action for state 1:
$:acc
a:
b:
Goto for state 1:
S:
A:
Action for state 2:
$:
a:s3
b:s4
Goto for state 2:
S:
A:5
Action for state 3:
$:
a:s3
b:s4
Goto for state 3:
S:
A:6
Action for state 4:
$:r3
a:r3
```




[in empty places, just press Enter(WHITE SPACE NOT ALLOWED)]

**EXP 8(IMPLEMENTING ADHOC-SDT TO CONVERT IF-ELSE TO SWITCH)**

*LEX code* :-

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

```
#include "y.tab.h"
extern int yylval;
extern int yylex();
extern int yyerror(const char *);
%}
%%
[ \t]                       ;
"if"                        return IF;
"else if"           return ELIF;
"else"                      return ELSE;
printf\(.*\)        return PRINTF;
[a-zA-Z%][a-zA-Z0-9+]*      return ID;
[0-9]+                      {yylval = atoi(yytext); return NUM;}
"=="                return EQ;
.                   return yytext[0];
%%
```

*YACC code* :-

```
%{
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
extern char *yytext;
int indent=0,i;
void indentation(int j);
void p1();
void p2();
void p3();
void p4();
void p5();
void p6();
%}
%token NUM IF ELSE ELIF PRINTF ID EQ
%start G

%%
G : S {printf("\n\nVALID PROG!\n");} ;
```

```
S : if elif ;
S2 : {indent+=2;} S | text ;
if : IF '(' ID{p1();} EQ NUM{p2();} ')' ST ;
elif : ELIF '(' ID EQ NUM{p3();} ')' ST elif | else end;
else : ELSE {p5();} ST | ;
end : {p6();indent-=2;} ;
ST : '{' S2 '}' | text ;
text : PRINTF{p4();} ';' ;
%%

int yyerror(const char *){
printf("Error!");
}

void main(){
yyparse();
}

void indentation(int j){
for(i=0;i<indent+j;i++)
printf("\t");
}

void p1(){
if (indent==0)
printf("\n\nTransformed Code:\n");
indentation(0);
printf("switch(%s) {\n",yytext);
}

void p2(){
indentation(1);
printf("case %s:",yytext);
}

void p3(){
indentation(2);
printf("break;\n");
indentation(1);
printf("case %s:",yytext);
```

```
}

void p4(){
indentation(2);
printf("%s;",yytext);}

void p5(){
indentation(2);
printf("break;\n");
indentation(1);
printf("default:");
}

void p6(){
printf("\n");
indentation(0);
printf("}");
}
```

*Sample input* :-

```
if (x == 5) {
    printf("x is 5");
} else if (x == 10) {
    printf("x is 10");
} else {
    printf("x is neither 5 nor 10");
}
```

**EXP 9(POSTFIX CONVERSION AND THREE ADDRESS CODE GENERATION)**

**i) POSTFIX CONVERSION**

*LEX code* :-

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"
```

```
extern int yyval;
extern char *yytext;
extern int yyerror(const char *);
%}

%%
[0-9]+ {yylval = atoi(yytext); return DIGIT;}
[a-zA-Z][a-zA-Z0-9]* return ID;
[ \t] ;
"\n" yyterminate();
. return yytext[0];
%%
```

*YACC code* :-

```
%{
#include <stdio.h>
#include <stdlib.h>
extern char *yytext;
void pID();
%}

%token DIGIT ID
%left '+' '-'
%left '*' '/'

%%
start : T {printf("\n");}
;
T : T '+' T {printf("+ ");}
| T '-' T {printf("- ");}
| T '*' T {printf("* ");}
| T '/' T {printf("/ ");}
| '(' T ')' { }
| DIGIT {printf("%d " , $1);}
| ID {pID();}
;
%%

int main(){
```

```
printf("\nEnter expression:");
yyparse();
return 0;
}

void pID(){
printf("%s " ,yytext);
}

int yyerror(const char *)
{printf("Syntax Error\n");}
```

**ii) GENERATING THREE ADDRESS CODE**

*LEX code* :-

```
%{
#include<stdio.h>
#include<stdlib.h>
#include "y.tab.h"
extern int yyval;
extern char *yytext;
extern int yyerror(const char *);
%}
ALPHA [A-Za-z]
DIGIT [0-9]
%%
[ \t] ;
{ALPHA}({ALPHA}|{DIGIT})* return ID;
{DIGIT}+ {yylval=atoi(yytext); return NUM;}
"\n" yyterminate();
. return yytext[0];
%%
```

*YACC code* :-

```
%{
#include<ctype.h>
#include<string.h>
extern char *yytext;
```

```
char st[100][25];
int top=0;
int tint=0;
void push();
void pushTemp();
void stack();
void codegen();
void codegen_umin();
void codegen_assign();
%}

%token ID NUM
%right '='
%left '+' '-'
%left '*' '/'
%left UMINUS
%%
S : ID{push();} '='{push();} E{codegen_assign();}
;
E : E '+'{push();} T{codegen();}
| E '-'{push();} T{codegen();}
| T
;
T : T '*'{push();} F{codegen();}
| T '/'{push();} F{codegen();}
| F
;
F : '(' E ')'
| '-'{push();} F{codegen_umin();} %prec UMINUS
| ID{push();}
| NUM{push();}
;
%%


void main()
{
printf("Enter the expression : ");
yyparse();
}
```

```
void push()
{
strcpy(st[++top],yytext);
}

void pushTemp(){
char X[]="t";
char Y[100];
sprintf(Y,"%d",tint);
strcpy(st[top],strcat(X,Y));
tint++;
}

void stack(){
int i;
printf("\nStack: ");
for (i=top;i>=0;i--)
printf("%s ",st[i]);
printf("\n");
}

void codegen()
{
stack();
printf("t%d = %s %s %s\n",tint,st[top-2],st[top-1],st[top]);
top-=2;
pushTemp();
}

void codegen_umin()
{
stack();
printf("t%d = -%s\n",tint,st[top]);
top--;
pushTemp();
}

void codegen_assign()
{
```

```
stack();
printf("%s = %s\n",st[top-2],st[top]);
top-=2;
}

int yyerror(const char *){
printf("Error");
}
```

**EXP 10(COMMON SUB EXPRESSION ELIMINATION)**

```
def optimizer(expressions):
    value_table = {}
    for i,(target,expr) in enumerate(expressions):
        if ' ' in expr:
            L,op,R = expr.split(' ')
            if(L.isdigit() and R.isdigit()):
                if(op=='+'):
                    expressions[i] = (target,int(L)+int(R))
                if(op=='-'):
                    expressions[i] = (target,int(L)-int(R))
                if(op=='*'):
                    expressions[i] = (target,int(L)*int(R))
                if(op=='/'):
                    expressions[i] = (target,int(L)/int(R))
            else:
                L_value = value_table.get(L,ord(L))
                R_value = value_table.get(R,ord(R))
                #creating hash_key
                hash_key = (op,min(L_value,R_value),max(L_value,R_value))
                #Check dictionary value
                if hash_key in value_table:
                    expressions[i] = (target,expressions[value_table[hash_key]][0])
                    value_table[hash_key] = i
                else:
                    value_table[hash_key] = i
        value_table[target]=i
    return expressions

expressions = [
```

```
        ('t1','x + y'),
        ('t2','x + y'),
        ('t3','8 + 5'),
        ('t4','x + z'),
        ('t5','6 - 3'),
        ('t5','5 * 3'),
    ]

ans = optimizer(expressions)

for target,expr in ans:
    print(f"{target} = {expr}")
```

[***Steps for giving input -*** DO NOT FORGET TO INCLUDE SPACE IF THE RHS OF EXPRESSION HAS TWO VARIABLES. ALSO IF RHS CONTAINS SINGLE VARIABLE DON'T INCLUDE ANY SPACE]

**EXP 11(CODE GENERATION)**

*LEX code* :-

```
%{
#include<stdio.h>
#include<string.h>
#include "y.tab.h"
extern char *yytext;
extern int yylval;
%}
%%
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* { return ID; }
[ \t] ;
\n return EOL;
. return yytext[0];
%%
```

*YACC code* :-

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

```
extern char *yytext;
void push();
void pop();
void p1();
void p2();
%}
%token NUMBER ID EOL
%%
G : S G | S;
S : ID{push();} '='{printf("LDA ");} E{pop();} EOL;
E : E '+'{printf("LDT ");} E{printf("ADDR A,T\n");}
| E '-'{printf("LDT ");} E{printf("SUBR A,T\n");}
| E '*'{printf("LDT ");} E{printf("MULR A,T\n");}
| E '/'{printf("LDT ");} E{printf("DIVR A,T\n");}
| NUMBER{p2();}
| ID{p1();}
;
%%

char x[100];

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

void push(){
strcpy(x,yytext);
}

void pop(){
printf("STA %s\n\n",x);
}

void p1(){
printf("%s\n",yytext);
}

void p2(){
printf("#%s\n",yytext);
```

```
}

int yyerror(const char *)
{
printf("Error!");
}
```