**VASAVI COLLEGE OF ENGINEERING (AUTONOMOUS)**

**IBRAHIMBAGH, HYDERABAD-31**

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**List of CC Lab External programs**

1. A) Implement Lexical analyzer using Lex

%{  
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()  
{  
yylex();  
printf("\n");  
return(0);  
}  
int yywrap()  
{  
return(1);  
}

B) Implement the SLR (1) parsing table for the given grammar

E->E+T|T

T->T\*F|T

F->id| (E)

import copy

def grammarAugmentation(rules, nonterm\_userdef,

start\_symbol):

newRules = []

newChar = start\_symbol + "'"

while (newChar in nonterm\_userdef):

newChar += "'"

newRules.append([newChar,

['.', start\_symbol]])

for rule in rules:

k = rule.split("->")

lhs = k[0].strip()

rhs = k[1].strip()

multirhs = rhs.split('|')

for rhs1 in multirhs:

rhs1 = rhs1.strip().split()

rhs1.insert(0, '.')

newRules.append([lhs, rhs1])

def generateStates(statesDict):

prev\_len = -1

called\_GOTO\_on = []

while (len(statesDict) != prev\_len):

prev\_len = len(statesDict)

keys = list(statesDict.keys())

for key in keys:

if key not in called\_GOTO\_on:

called\_GOTO\_on.append(key)

compute\_GOTO(key)

return

def first(rule):

global rules, nonterm\_userdef, \

term\_userdef, diction, firsts

if len(rule) != 0 and (rule is not None):

if rule[0] in term\_userdef:

return rule[0]

elif rule[0] == '#':

return '#'

if len(rule) != 0:

if rule[0] in list(diction.keys()):

fres = []

rhs\_rules = diction[rule[0]]

for itr in rhs\_rules:

indivRes = first(itr)

if type(indivRes) is list:

for i in indivRes:

fres.append(i)

else:

fres.append(indivRes)

if '#' not in fres:

return fres

else:

newList = []

fres.remove('#')

if len(rule) > 1:

ansNew = first(rule[1:])

if ansNew != None:

if type(ansNew) is list:

newList = fres + ansNew

else:

newList = fres + [ansNew]

else:

newList = fres

return newList

if res is not None:

if type(res) is list:

for g in res:

solset.add(g)

else:

ptr = 0

j = 0

for y in Table:

frmt1 = "{:>8}" \* len(y)

print(f"{{:>3}} {frmt1.format(\*y)}".format('I'+str(j)))

j += 1

def printResult(rules):

for rule in rules:

print(f"{rule[0]} ->"

f" {' '.join(rule[1])}")

def printAllGOTO(diction):

for itr in diction:

print(f"GOTO ( I{itr[0]} ,"f" {itr[1]} ) = I{stateMap[itr]}")

rules = ["E -> E + T | T",

"T -> T \* F | F",

"F -> ( E ) | id"]

nonterm\_userdef = ['E', 'T', 'F']

term\_userdef = ['id', '+', '\*', '(', ')']

start\_symbol = nonterm\_userdef[0]

print("\nOriginal grammar input:\n")

for y in rules:

print(y)

print("\nGrammar after Augmentation: \n")

rules = \grammarAugmentation(rules,nonterm\_userdef,start\_symbol)

printResult(rules)

start\_symbol = rules[0][0]

print("\nCalculated closure: I0\n")

I0 = findClosure(0, start\_symbol)

printResult(I0)

statesDict = {}

stateMap = {}

statesDict[0] = I0

stateCount = 0

generateStates(statesDict)

print("\nStates Generated: \n")

for st in statesDict:

print(f"State = I{st}")

printResult(statesDict[st])

print()

print("Result of GOTO computation:\n")

printAllGOTO(stateMap)

diction = {}

createParseTable(statesDict, stateMap,term\_userdef,nonterm\_userdef)

1. A) Lex Program to recognize the numbers which has 1 in its 5th position from right**.**

%%

[1-9]\*1[1-9]{4} {printf(“satisfying”);}

%%

main()

{

yylex();

}

int yywrap()

{

return 1;

}

1. Construct target code for the given expression**.**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int label[20],no=0;

int main(){

FILE \*fp1,\*fp2;

int check\_label(int n);

char fname[100],op[10],ch;

char op1[8],op2[8],res[8];

int i=0; int j=0;

printf("\n enter filename of intermediate code:");

scanf("%s",fname);// printf("%s",fname);

fp1=fopen(fname,"r");

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

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

printf("\n error in opening files....");

exit(0);

}

while(!feof(fp1)){

fprintf(fp2,"\n");

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

i++;

if(check\_label(i))

fprintf(fp2,"\n label.# %d:",i);

if(strcmp(op,"printf")==0){

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

fprintf(fp2,"\n\t OUT %s",res);

}

if(strcmp(op,"goto")==0){

fscanf(fp1,"%s%s",op1,op2);

fprintf(fp2,"\n\t JMP %s label.# %s",op1,op2);

label[no++]=atoi(op2);

}

if(strcmp(op,"[]=")==0){

fscanf(fp1,"%s%s%s",op1,op2,res);

fprintf(fp2,"\n\t STORE %s[%s],%s",op1,op2,res);

}

if(strcmp(op,"uminus")==0){

fscanf(fp1,"%s%s",op1,res);

fprintf(fp2,"\n\t LOAD %s, R1",op1);

fprintf(fp2,"\n\t STORE R1,%s",res);

}

switch(op[0]){

case '\*': fscanf(fp1,"%s%s%s",op1,op2,res);

fprintf(fp2,"\n\t LOAD %s,R0",op1);

fprintf(fp2,"\n\t LOAD %s,R1",op2);

fprintf(fp2,"\n\t MUL R1,R0");

fprintf(fp2,"\n\t STORE R0,%s",res);

break;

case '+': fscanf(fp1,"%s%s%s",op1,op2,res);

fprintf(fp2,"\n\t LOAD %s,R0",op1);

fprintf(fp2,"\n\t LOAD %s,R1",op2);

fprintf(fp2,"\n\t ADD R1,R0");

fprintf(fp2,"\n\t STORE R0,%s",res);

break;

case '-': fscanf(fp1,"%s%s%s",op1,op2,res);

fprintf(fp2,"\n\t LOAD %s,R0",op1);

fprintf(fp2,"\n\t LOAD %s,R1",op2);

fprintf(fp2,"\n\t SUB R1,R0");

fprintf(fp2,"\n\t STORE R0,%s",res);

break;

case '/': fscanf(fp1,"%s%s%s",op1,op2,res);

fprintf(fp2,"\n\t LOAD %s,R0",op1);

fprintf(fp2,"\n\t LOAD %s,R!",op2);

fprintf(fp2,"\n\t DIV R1,R0");

fprintf(fp2,"\n\t STORE R0,%s",res);

break;

case '%': fscanf(fp1,"%s%s%s",op1,op2,res);

fprintf(fp2,"\n\t LOAD %s,R0",op1);

fprintf(fp2,"\n\t LOAD %s,R1",op2);

fprintf(fp2,"\n\t DIV R1,R0");

fprintf(fp2,"\n\t STORE R0,%s",res);

break;

case '=': fscanf(fp1,"%s%s",op1,res);

fprintf(fp2,"\n\t STORE %s, %s",op1,res);

break;

case '>': j++;fscanf(fp1,"%s%s%s",op1,op2,res);

fprintf(fp2,"\n\t LOAD %s,R0",op1);

fprintf(fp2,"\n\t JGT %s,label.# %s",op2,res);

label[no++]=atoi(res);

break;

}

}

fclose(fp2);

fclose(fp1);

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

if(fp2==NULL){

printf("\n error in opening file target.txt");

exit(0);

}

do{

ch=fgetc(fp2);

printf("%c",ch);

}while(ch!=EOF);

fclose(fp2);

return 0;

}

int check\_label(int k){

int i;for(i=0;i<no;i++){

if(k==label[i])

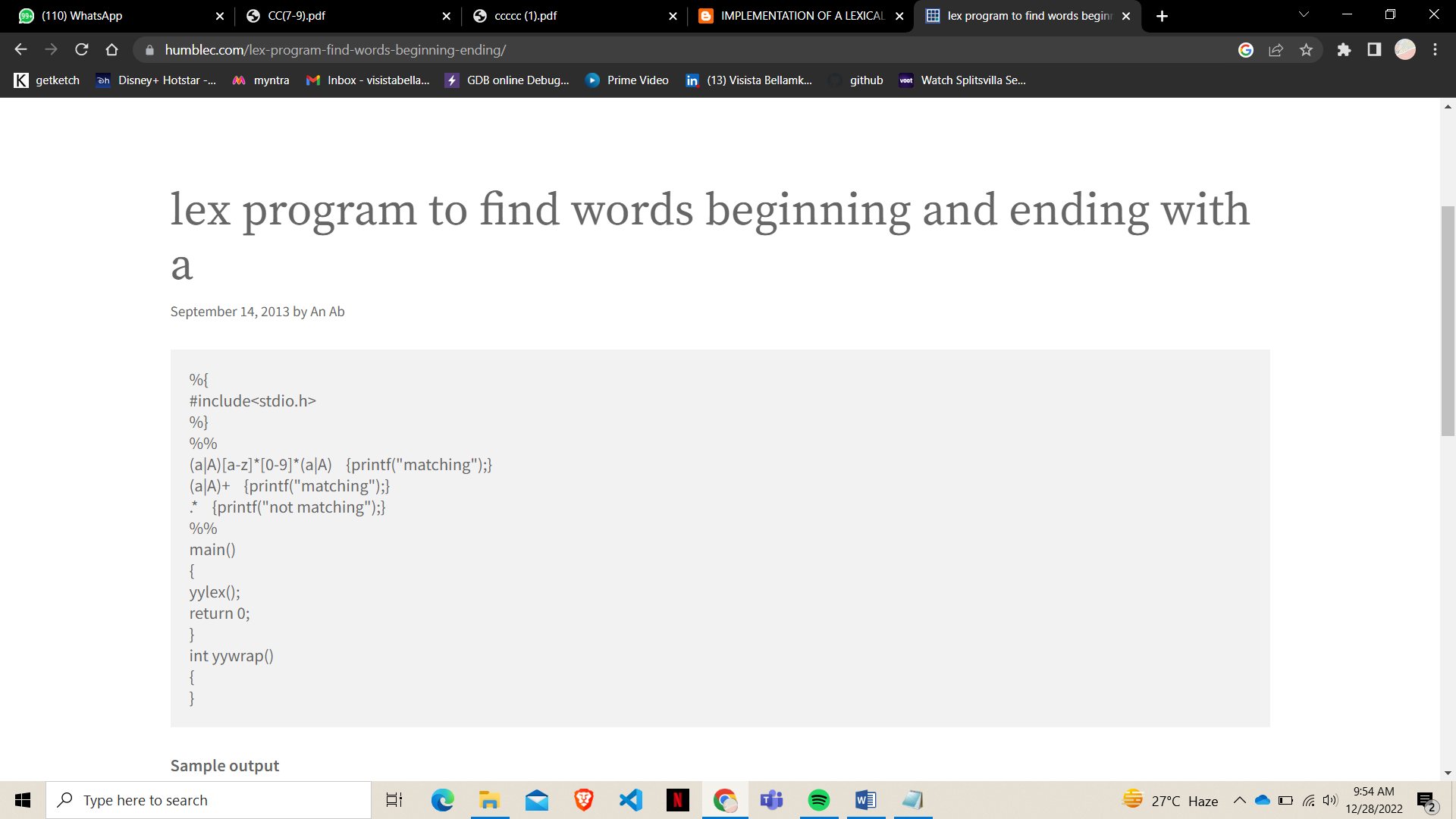
return 1;

}

return 0;

}

1. A) Lex Program to recognize the strings which are starting or ending with ‘a’



B) Implement the Three address code using YACC

three.l

%{

#include "y.tab.h"

extern char yyval;

%}

number [0-9]+

letter [a-zA-Z]+

%%

{number} {yylval.sym=(char)yytext[0];return number;}

{letter} {yylval.sym=(char)yytext[0]; return letter; }

\n {return 0;}

. {return yytext[0];}

%%

Three.y

%{

#include<stdio.h>

#include<string.h>

int nIndex=0;

struct Intercode

{

char operand1;

char operand2;

char opera;

};

%}

%union

{

char sym;

}

%token <sym> letter number

%type <sym> expr

%left '-' '+'

%right '\*' '/'

%%

statement: letter '=' expr ';' { addtotable((char)$1,(char)$3,'=' ); }

| expr ;

;

expr: expr '+' expr { $$=addtotable((char)$1,(char)$3,'+');}

| expr '-' expr { $$=addtotable((char)$1,(char)$3,'-');}

| expr '\*' expr { $$=addtotable((char)$1,(char)$3, '\*');}

| expr '/' expr { $$=addtotable((char)$1,(char)$3,'/');}

| '(' expr ')' { $$= (char)$2;}

| number { $$= (char)$1;}

| letter { $$= (char)$1;}

%%

yyerror(char \*s)

{

printf("%s",s);

exit (0);

}

struct Intercode code[20];

char temp = 'A';

int f=0;

char addtotable(char operand1, char operand2,char opera)

{

if(f!=0)

temp++;

code[nIndex].operand1 = operand1;

code[nIndex].operand2 = operand2;

code[nIndex].opera = opera;

nIndex++;

f++;

return temp;

}

threeaddresscode()

{

int nCnt=0;

char temp='A';

printf("\n\n\t three address codes\n\n");

while(nCnt<nIndex)

{

printf("%c:=\t",temp);

if (isalpha(code[nCnt].operand1))

printf("%c\t", code[nCnt].operand1);

else

printf("%c\t",temp);

printf("%c\t", code[nCnt].opera);

if (isalpha(code[nCnt].operand2))

printf("%c\t", code[nCnt].operand2);

else

printf("%c\t",temp);

printf("\n");

nCnt++;

temp++;

}}

main()

{

printf("enter expression");

yyparse();

threeaddresscode();

}

yywrap()

{

return 1;

}

1. A) Lex program to recognize String ending with 11.

%%

[0-9]\*11{printf(“string accepted”);

[0-9]\*{printf(“string rejected”);}

%%

main()

{

yylex();

}

int yywrap()

{

return 1;

}

B) Construct DAG for the given three address code

#include<stdio.h>

#include<ctype h>

#define size 20

typedef struct node

{

char data;

struct node \*left;

struct node \*right;

}btree;

btree \*stack[size];

int top;

main()

{

btree \*root; char exp[80];

btree \*create(char exp[80]);

void dag(btree \*root);

printf("\nEnter the postfix expression:\n");

scanf("%s",exp);

top=-1;

root=create(exp);

printf("\nThe tree is created.....\n");

printf("\nInorder DAG is : \n\n");

dag(root);

return 0;

}

btree \*create(char exp[])

{

btree \*temp; int pos; char ch;

void push(btree\*);

btree \*pop();

pos=0;

ch=exp[pos];

printf("%c\t",ch);

while(ch!='\0')

{

temp=((btree\*)malloc(sizeof(btree)));

temp->left=temp->right=NULL;

temp->data=ch;

printf("%c",temp->data);

if(isalpha(ch))

push(temp);

else if(ch=='+' ||ch=='-' || ch=='\*' || ch=='/')

{

temp->right=pop();

temp->left=pop();

push(temp);

}

else

printf("\n Invalid char Expression\n");

pos++;

ch=exp[pos];

}

temp=pop();

return(temp);

}

void push(btree \*Node)

{

if(top+1 >=size)

printf("Error:Stack is full\n");

top++;

stack[top]=Node;

}

btree\* pop()

{

btree \*Node;

if(top==-1)

printf("\nerror: stack is empty..\n");

Node=stack[top];

top--;

return(Node);

}

void dag(btree \*root)

{

btree \*temp;

temp=root;

if(temp!=NULL)

{

dag(temp->left); printf("%c",temp->data); dag(temp->right);

}

}

1. A) Lex program to recognize Keywords

%{

#include <stdio.h>

%}

%%

If | else | while | int | switch | for | char (printf("keyword");}

.\* {(printf("invalid");}

%%

main(){

yylex();

return 0;

}

int yywrap();

{}

B) Implement the Dependency graph

1. A) Lex program to recognize String ending with 00.

%%

[0-9]\*00{printf(“string accepted”);

[0-9]\*{printf(“string rejected”);}

%%

main()

{

yylex();

}

int yywrap()

{

return 1;

}

B) Implement the CLR parsing table for the given grammar

S->CC

C->aC|d

**firstfollow.py:**

from re import \*

from collections import OrderedDict

t\_list=OrderedDict()

nt\_list=OrderedDict()

production\_list=[]

class Terminal:

def \_\_init\_\_(self, symbol):

self.symbol=symbol

def \_\_str\_\_(self):

return self.symbol

class NonTerminal:

def \_\_init\_\_(self, symbol):

self.symbol=symbol

self.first=set()

self.follow=set()

def \_\_str\_\_(self):

return self.symbol

def add\_first(self, symbols): self.first |= set(symbols)

def add\_follow(self, symbols): self.follow |= set(symbols)

def compute\_first(symbol): #chr(1013) corresponds (ϵ) in Unicode

global production\_list, nt\_list, t\_list

if symbol in t\_list:

return set(symbol)

for prod in production\_list:

head, body=prod.split('->')

if head!=symbol: continue

if body=='':

nt\_list[symbol].add\_first(chr(1013))

continue

for i, Y in enumerate(body):

if body[i]==symbol: continue

t=compute\_first(Y)

nt\_list[symbol].add\_first(t-set(chr(1013)))

if chr(1013) not in t:

break

if i==len(body)-1:

nt\_list[symbol].add\_first(chr(1013))

return nt\_list[symbol].first

def get\_first(symbol): #wrapper method for compute\_first

return compute\_first(symbol)

def compute\_follow(symbol):

global production\_list, nt\_list, t\_list

if symbol == list(nt\_list.keys())[0]:

nt\_list[symbol].add\_follow('$')

for prod in production\_list:

head, body=prod.split('->')

for i, B in enumerate(body):

if B != symbol: continue

if i != len(body)-1:

nt\_list[symbol].add\_follow(get\_first(body[i+1]) - set(chr(1013)))

if i == len(body)-1 or chr(1013) in get\_first(body[i+1]) and B != head:

nt\_list[symbol].add\_follow(get\_follow(head))

def get\_follow(symbol):

global nt\_list, t\_list

if symbol in t\_list.keys():

return None

return nt\_list[symbol].follow

def main(pl=None):

print('''Enter the grammar productions (enter 'end' or return to stop)

global production\_list, t\_list, nt\_list

ctr=1

if pl==None:

while True:

production\_list.append(input().replace(' ', ''))

if production\_list[-1].lower() in ['end', '']:

del production\_list[-1]

break

head, body=production\_list[ctr-1].split('->')

if head not in nt\_list.keys():

nt\_list[head]=NonTerminal(head)

for i in body:

if not 65<=ord(i)<=90:

if i not in t\_list.keys(): t\_list[i]=Terminal(i)

elif i not in nt\_list.keys(): nt\_list[i]=NonTerminal(i)

ctr+=1

return pl

if \_\_name\_\_=='\_\_main\_\_':

main()

**clr.py:**

from collections import deque

from collections import OrderedDict

from pprint import pprint

import firstfollow

from firstfollow import production\_list, nt\_list as ntl, t\_list as tl

nt\_list, t\_list=[], []

class State:

\_id=0

def \_\_init\_\_(self, closure):

self.closure=closure

self.no=State.\_id

State.\_id+=1

class Item(str):

def \_\_new\_\_(cls, item, lookahead=list()):

self=str.\_\_new\_\_(cls, item)

self.lookahead=lookahead

return self

def \_\_str\_\_(self):

return super(Item, self).\_\_str\_\_()+", "+'|'.join(self.lookahead)

def closure(items):

def exists(newitem, items):

for i in items:

if i==newitem and sorted(set(i.lookahead)) == sorted(set(newitem.lookahead)):

return True

return False

global production\_list

while True:

flag=0

for i in items:

if i.index('.')==len(i)-1: continue

Y=i.split('->')[1].split('.')[1][0]

if i.index('.')+1<len(i)-1:

lastr = list(firstfollow.compute\_first(i[i.index('.')+2])-set(chr(1013)))

else:

lastr=i.lookahead

for prod in production\_list:

head, body=prod.split('->')

if head!=Y: continue

newitem=Item(Y+'->.'+body, lastr)

if not exists(newitem, items):

items.append(newitem)

flag=1

if flag==0: break

return items

def goto(items, symbol):

global production\_list

initial=[]

for i in items:

if i.index('.')==len(i)-1: continue

head, body=i.split('->')

seen, unseen=body.split('.')

if unseen[0]==symbol and len(unseen) >= 1:

initial.append(Item(head+'->'+seen+unseen[0]+'.'+unseen[1:], i.lookahead))

return closure(initial)

def calc\_states():

def contains(states, t):

for s in states:

if len(s) != len(t): continue

if sorted(s)==sorted(t):

for i in range(len(s)):

if s[i].lookahead!=t[i].lookahead: break

else: return True

return False

global production\_list, nt\_list, t\_list

head, body=production\_list[0].split('->')

states=[closure([Item(head+'->.'+body, ['$'])])]

while True:

flag=0

for s in states:

for e in nt\_list+t\_list:

t=goto(s, e)

if t == [] or contains(states, t): continue

states.append(t)

flag=1

if not flag: break

return states

def make\_table(states):

global nt\_list, t\_list

def getstateno(t):

for s in states:

if len(s.closure) != len(t): continue

if sorted(s.closure)==sorted(t):

for i in range(len(s.closure)):

if s.closure[i].lookahead!=t[i].lookahead: break

else: return s.no

return -1

def getprodno(closure):

closure=''.join(closure).replace('.', '')

return production\_list.index(closure)

SLR\_Table=OrderedDict()

for i in range(len(states)):

states[i]=State(states[i])

for s in states:

SLR\_Table[s.no]=OrderedDict()

for item in s.closure:

head, body=item.split('->')

if body=='.':

for term in item.lookahead:

if term not in SLR\_Table[s.no].keys():

SLR\_Table[s.no][term]={'r'+str(getprodno(item))}

else: SLR\_Table[s.no][term] |= {'r'+str(getprodno(item))}

continue

nextsym=body.split('.')[1]

if nextsym=='':

if getprodno(item)==0:

SLR\_Table[s.no]['$']='accept'

else:

for term in item.lookahead:

if term not in SLR\_Table[s.no].keys():

SLR\_Table[s.no][term]={'r'+str(getprodno(item))}

else: SLR\_Table[s.no][term] |= {'r'+str(getprodno(item))}

continue

nextsym=nextsym[0]

t=goto(s.closure, nextsym)

if t != []:

if nextsym in t\_list:

if nextsym not in SLR\_Table[s.no].keys():

SLR\_Table[s.no][nextsym]={'s'+str(getstateno(t))}

else: SLR\_Table[s.no][nextsym] |= {'s'+str(getstateno(t))}

else: SLR\_Table[s.no][nextsym] = str(getstateno(t))

return SLR\_Table

def augment\_grammar():

for i in range(ord('Z'), ord('A')-1, -1):

print(b[0][0])

print(a[0][1]["S"])'''

print("productions\t:",production\_list)

print('stack',"\t \t\t \t",'Input')

print(\*stack,"\t \t\t \t",\*Input,sep="")

while(len(Input)!=0):

b=list(a[int(stack[-1])][1][Input[0]])

if(b[0][0]=="s" ):

stack.append(Input[0])

stack.append(b[0][1:])

Input=Input[1:]

print(\*stack,"\t \t\t \t",\*Input,sep="")

elif(b[0][0]=="r" ):

s=int(b[0][1:])

l=len(production\_list[s])-3

prod=production\_list[s]

l\*=2

l=len(stack)-l

stack=stack[:l]

s=a[int(stack[-1])][1][prod[0]]

stack+=list(prod[0])

stack.append(s)

print(\*stack,"\t \t\t \t",\*Input,sep="")

elif(b[0][0]=="a"):

print("\n\tString Accepted\n")

break

except:

print('\n\tString INCORRECT for given Grammar!\n')

return

if \_\_name\_\_=="\_\_main\_\_":

main()

1. A) Lex program to recognize Keywords

B) Implement Intermediate code generation using YACC

1. A) Lex program to assign line numbers for source code

%{

int line\_number = 1;

%}

%%

{line} { printf("%10d %s", line\_number++, yytext); }

%%

int yywrap(){}

int main(int argc, char\*argv[])

{

extern FILE \*yyin;

yyin = fopen("testtest.c","r");

yylex();

return 0;

}

B) Implement First & Follow

#include<stdio.h>

#include<ctype.h>

#include<string.h>

void followfirst(char, int, int);

void follow(char c);

void findfirst(char, int, int);

int count, n = 0;

char calc\_first[10][100];

char calc\_follow[10][100];

int m = 0;

char production[10][10];

char f[10], first[10];

int k,e;

char ck;

int main(int argc, char \*\*argv)

{

int jm = 0, km=0;

int i, choice;

char c, ch;

count = 8;

strcpy(production[0], "E=TR");

strcpy(production[1], "R=+TR");

strcpy(production[2], "R=#");

strcpy(production[3], "T=FY");

strcpy(production[4], "Y=\*FY");

strcpy(production[5], "Y=#");

strcpy(production[6], "F=(E)");

strcpy(production[7], "F=i");

int kay;

char done[count];

int ptr = -1;

for(k = 0; k < count; k++) {

for(kay = 0; kay < 100; kay++) {

calc\_first[k][kay] = '!';

}

}

int point1 = 0, point2, xxx;

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

{

c = production[k][0];

point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(c == done[kay])

xxx = 1;

if (xxx == 1)

continue;

findfirst(c, 0, 0);

ptr += 1;

done[ptr] = c;

printf("\n First(%c) = { ", c);

calc\_first[point1][point2++] = c;

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

int lark = 0, chk = 0;

for(lark = 0; lark < point2; lark++) {

if (first[i] == calc\_first[point1][lark])

{

chk = 1;

break;

}

}

if(chk == 0)

{

printf("%c, ", first[i]);

calc\_first[point1][point2++] = first[i];

}

}

printf("}\n");

jm = n;

point1++;

}

printf("\n");

char donee[count];

ptr = -1;

for(k = 0; k < count; k++) {

for(kay = 0; kay < 100; kay++) {

calc\_follow[k][kay] = '!';

}

}

point1 = 0;

int land = 0;

for(e = 0; e < count; e++)

{

ck = production[e][0];

point2 = 0;

xxx = 0;

for(kay = 0; kay <= ptr; kay++)

if(ck == donee[kay])

xxx = 1;

if (xxx == 1)

continue;

land += 1;

follow(ck);

ptr += 1;

donee[ptr] = ck;

printf(" Follow(%c) = { ", ck);

calc\_follow[point1][point2++] = ck;

for(i = 0 + km; i < m; i++) {

int lark = 0, chk = 0;

for(lark = 0; lark < point2; lark++)

{

if (f[i] == calc\_follow[point1][lark])

{

chk = 1;

break;

}

}

if(chk == 0)

{

printf("%c, ", f[i]);

calc\_follow[point1][point2++] = f[i];

}

}

printf(" }\n\n");

km = m;

point1++;

}

}

void follow(char c)

{

int i, j;

if(production[0][0] == c) {

f[m++] = '$';

}

void findfirst(char c, int q1, int q2)

{

int j;

if(!(isupper(c))) {

first[n++] = c;

}

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

{

if(production[j][0] == c)

{

if(production[j][2] == '#')

{

if(production[q1][q2] == '\0')

first[n++] = '#';

else if(production[q1][q2] != '\0'

&& (q1 != 0 || q2 != 0))

{

findfirst(production[q1][q2], q1, (q2+1));

}

else

first[n++] = '#';

}

else if(!isupper(production[j][2]))

{

first[n++] = production[j][2];

}

else

{

findfirst(production[j][2], j, 3);

}}}}

void followfirst(char c, int c1, int c2)

{

int k;

if(!(isupper(c)))

f[m++] = c;

else

{

int i = 0, j = 1;

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

{

if(calc\_first[i][0] == c)

break;

}

j++;

}}}

1. A) Lex program to recognize Identifiers.

%{

#include <stdio.h>

%}

%%

[a-z]([a-z]|[0-9]) (printf("identifier");}

.\* {(printf("invalid");}

%%

main(){

yylex();

return 0;

}

int yywrap();

{}

B) Implement a YACC specification for simple arithmetic calculations.

Lex program:

%{

#include "y.tab.h"

#include<math.h>

%}

%%

([0-9]+|([0-9]\*\.[0-9]+)([eE][-+]?[0-9]+)?) {yylval.dval=atof(yytext);

return NUMBER;

}

log|LOG {return LOG;}

ln {return nLOG;}

sin|SIN {return SINE;}

cos|COS {return COS;}

tan|TAN {return TAN;}

mem {return MEM;}

[\t];

\$; {return 0;}

\n|. {return yytext[0];}

%%

yacc program

%{

#include<stdio.h>

#include<math.h>

double memvar;

%}

%union

{

double dval;

}

%token<dval>NUMBER

%token<dval>MEM

%token LOG SINE nLOG COS TAN

%left '-''+'

%left '\*''/'

%right '^'

%left LOG SINE nLOG COS TAN

%nonassoc UMINUS

%type<dval> expression

%%

start: statement '\n'

|start statement '\n'

;

statement: MEM'='expression { memvar=$3;}

|expression {printf("answer=%g\n",$1);}

;

expression:expression'+'expression {$$=$1+$3;}

|expression'-'expression {$$=$1+$3;}

|expression'\*'expression {$$=$1\*$3;}

|expression'/'expression

{

if($3==0)

yyerror("divide by zero");

else

$$=$1/$3;}

|expression'^'expression {$$=pow($1,$3);}

;

expression: '-' expression %prec UMINUS {$$=-$2;}

|'('expression')' {$$=$2;}

|LOG expression {$$=log($2)/log(10);}

|nLOG expression {$$=log($2);}

|SINE expression {$$=sin($2\*3.14159/180);}

|COS expression {$$=cos($2\*3.14159/180);}

|TAN expression {$$=tan($2\*3.14159/180);}

|NUMBER { $$ = $1;}

|MEM {$$=memvar;}

;

%%

main()

{

printf("enter expression:");

yyparse();

}

int yyerror(char \*error)

{

fprintf(stderr,"%s\n",error);

}

yywrap()

{ return 1;

}

1. A) Lex program to recognize operators

%{

#include <stdio.h>

%}

%%

== | > | < | = | <= | >= | != {(printf("operator");}

.\* {(printf("invalid");}

%%

main(){

yylex();

return 0;

}

int yywrap();

{}

B) Implement LL (1) Parser.

#include<stdio.h>

#include<string.h>

#define TSIZE 128

int table[100][TSIZE];

char terminal[TSIZE];

nonterminal is the grammar

char nonterminal[26];

struct product {

char str[100];

int len;

}pro[20];

int no\_pro;

char first[26][TSIZE];

char follow[26][TSIZE];

char first\_rhs[100][TSIZE];

int isNT(char c) {

return c >= 'A' && c <= 'Z';

}

void readFromFile() {

FILE\* fptr;

fptr = fopen("text.txt", "r");

char buffer[255];

int i;

int j;

while (fgets(buffer, sizeof(buffer), fptr)) {

printf("%s", buffer);

j = 0;

nonterminal[buffer[0] - 'A'] = 1;

for (i = 0; i < strlen(buffer) - 1; ++i) {

if (buffer[i] == '|') {

++no\_pro;

pro[no\_pro - 1].str[j] = '\0';

pro[no\_pro - 1].len = j;

pro[no\_pro].str[0] = pro[no\_pro - 1].str[0];

pro[no\_pro].str[1] = pro[no\_pro - 1].str[1];

pro[no\_pro].str[2] = pro[no\_pro - 1].str[2];

j = 3;

}

else {

pro[no\_pro].str[j] = buffer[i];

++j;

if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') {

terminal[buffer[i]] = 1;

}

}

}

pro[no\_pro].len = j;

++no\_pro;

}

}

void add\_FIRST\_A\_to\_FOLLOW\_B(char A, char B) {

int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^')

follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];

}

}

void add\_FOLLOW\_A\_to\_FOLLOW\_B(char A, char B) {

int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^')

follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];

}

}

void FOLLOW() {

int t = 0;

int i, j, k, x;

while (t++ < no\_pro) {

for (k = 0; k < 26; ++k) {

if (!nonterminal[k]) continue;

char nt = k + 'A';

for (i = 0; i < no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) {

if (nt == pro[i].str[j]) {

for (x = j + 1; x < pro[i].len; ++x) {

char sc = pro[i].str[x];

if (isNT(sc)) {

add\_FIRST\_A\_to\_FOLLOW\_B(sc, nt);

if (first[sc - 'A']['^'])

continue;

}

else {

follow[nt - 'A'][sc] = 1;

}

break;

}

if (x == pro[i].len)

add\_FOLLOW\_A\_to\_FOLLOW\_B(pro[i].str[0], nt);

}}}}}}

void add\_FIRST\_A\_to\_FIRST\_B(char A, char B) {

int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^') {

first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];

}}}

void FIRST() {

int i, j;

int t = 0;

while (t < no\_pro) {

for (i = 0; i < no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) {

char sc = pro[i].str[j];

if (isNT(sc)) {

add\_FIRST\_A\_to\_FIRST\_B(sc, pro[i].str[0]);

if (first[sc - 'A']['^'])

continue;

}

else {

first[pro[i].str[0] - 'A'][sc] = 1;

}

break;

}

if (j == pro[i].len)

first[pro[i].str[0] - 'A']['^'] = 1;

}

++t;

}

}

void add\_FIRST\_A\_to\_FIRST\_RHS\_\_B(char A, int B) {

int i;

for (i = 0; i < TSIZE; ++i) {

if (i != '^')

first\_rhs[B][i] = first[A - 'A'][i] || first\_rhs[B][i];

}}

void FIRST\_RHS() {

int i, j;

int t = 0;

while (t < no\_pro) {

for (i = 0; i < no\_pro; ++i) {

for (j = 3; j < pro[i].len; ++j) {

char sc = pro[i].str[j];

if (isNT(sc)) {

add\_FIRST\_A\_to\_FIRST\_RHS\_\_B(sc, i);

if (first[sc - 'A']['^'])

continue;

}

else {

first\_rhs[i][sc] = 1;

}

break;

}

if (j == pro[i].len)

first\_rhs[i]['^'] = 1;

}

++t;

}

}

int main() {

readFromFile();

follow[pro[0].str[0] - 'A']['$'] = 1;

FIRST();

FOLLOW();

if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))

p = p + 1;

for (j = 0; j < TSIZE; ++j) {

if (first\_rhs[i][j] && j != '^') {

table[p][j] = i + 1;

}

else if (first\_rhs[i]['^']) {

for (k = 0; k < TSIZE; ++k) {

if (follow[pro[i].str[0] - 'A'][k]) {

table[p][k] = i + 1;

}}}}}

k = 0;

for (i = 0; i < no\_pro; ++i) {

if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {

printf("%-10c", pro[i].str[0]);

for (j = 0; j < TSIZE; ++j) {

if (table[k][j]) {

printf("%-10s", pro[table[k][j] - 1].str);

}

else if (terminal[j]) {

printf("%-10s", "");

}

}

++k;

printf("\n");

}}}

🡺recursive decent parsing

#include<stdio.h>

#include<string.h>

int E(),Edash(),T(),Tdash(),F();

char \*ip;

char string[50];

int main()

{

printf("Enter the string\n");

scanf("%s",string);

ip=string;

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

if(E() && ip=="\0"){

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

printf("\n String is successfully parsed\n");

}

else{

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

printf("Error in parsing String\n");

}

}

int E()

{

printf("%s\tE->TE' \n",ip);

if(T())

{

if(Edash()){

return 1;

}

else

return 0;

}

else

return 0;

}

int Edash(){

if(\*ip=='+'){

printf("%s\tE'->+TE' \n",ip);

ip++;

if(T()){

if(Edash()){

return 1;

}

else

return 0;

}

else

return 0;

}

else{

printf("%s\tE'->^ \n",ip);

return 1;

}

}

int T(){

printf("%s\tT->FT' \n",ip);

if(F()){

if(Tdash())

{

return 1;

}

else

return 0;

}

int F(){

if(\*ip=='('){

printf("%s\tF->(E) \n",ip);

ip++;

if(E()){

if(\*ip==')'){

ip++;

return 0;

}

else

return 0;

}

else

return 0;

}

else if(\*ip=='i'){

ip++;

printf("%s\tF->id \n",ip);

return 1;

}

else

return 0;

}