CC INTERNAL-I

1. Implement Lexical analyzer / Scanner using C.

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

// Returns 'true' if the character is a DELIMITER.
bool isDelimiter(char ch)
{
	if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
		ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
		ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
		ch == '[' || ch == ']' || ch == '{' || ch == '}')
		return (true);
	return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
	if (ch == '+' || ch == '-' || ch == '*' ||
		ch == '/' || ch == '>' || ch == '<' ||
		ch == '=')
		return (true);
	return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
{
	if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
		str[0] == '3' || str[0] == '4' || str[0] == '5' ||
		str[0] == '6' || str[0] == '7' || str[0] == '8' ||
		str[0] == '9' || isDelimiter(str[0]) == true)
		return (false);
	return (true);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
	if (!strcmp(str, "if") || !strcmp(str, "else") ||
		!strcmp(str, "while") || !strcmp(str, "do") ||
		!strcmp(str, "break") ||
		!strcmp(str, "continue") || !strcmp(str, "int")
		|| !strcmp(str, "double") || !strcmp(str, "float")
		|| !strcmp(str, "return") || !strcmp(str, "char")
		|| !strcmp(str, "case") || !strcmp(str, "char")
		|| !strcmp(str, "sizeof") || !strcmp(str, "long")
		|| !strcmp(str, "short") || !strcmp(str, "typedef")
		|| !strcmp(str, "switch") || !strcmp(str, "unsigned")
		|| !strcmp(str, "void") || !strcmp(str, "static")
		|| !strcmp(str, "struct") || !strcmp(str, "goto"))
		return (true);
	return (false);
}

// Returns 'true' if the string is an INTEGER.
```

```
bool isInteger(char* str)
{
	int i, len = strlen(str);

	if (len == 0)
		return (false);
	for (i = 0; i < len; i++) {
		if (str[i] != '0' && str[i] != '1' && str[i] != '2'
			&& str[i] != '3' && str[i] != '4' && str[i] != '5'
			&& str[i] != '6' && str[i] != '7' && str[i] != '8'
			&& str[i] != '9' || (str[i] == '-' && i > 0))
			return (false);
	}
	return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char* str)
{
	int i, len = strlen(str);
	bool hasDecimal = false;

	if (len == 0)
		return (false);
	for (i = 0; i < len; i++) {
		if (str[i] != '0' && str[i] != '1' && str[i] != '2'
			&& str[i] != '3' && str[i] != '4' && str[i] != '5'
			&& str[i] != '6' && str[i] != '7' && str[i] != '8'
			&& str[i] != '9' && str[i] != '.' ||
			(str[i] == '-' && i > 0))
			return (false);
		if (str[i] == '.')
			hasDecimal = true;
	}
	return (hasDecimal);
}

// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
	int i;
	char* subStr = (char*)malloc(
				sizeof(char) * (right - left + 2));

	for (i = left; i <= right; i++)
		subStr[i - left] = str[i];
	subStr[right - left + 1] = '\0';
	return (subStr);
}

// Parsing the input STRING.
void parse(char* str)
{
	int left = 0, right = 0;
	int len = strlen(str);

	while (right <= len && left <= right) {
		if (isDelimiter(str[right]) == false)
			right++;

		if (isDelimiter(str[right]) == true && left == right) {
			if (isOperator(str[right]) == true)
```

```
                printf("'%c' IS AN OPERATOR\n", str[right]);

            right++;
            left = right;
        } else if (isDelimiter(str[right]) == true && left != right
                || (right == len && left != right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("'%s' IS A KEYWORD\n", subStr);

            else if (isInteger(subStr) == true)
                printf("'%s' IS AN INTEGER\n", subStr);

            else if (isRealNumber(subStr) == true)
                printf("'%s' IS A REAL NUMBER\n", subStr);

            else if (validIdentifier(subStr) == true
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS A VALID IDENTIFIER\n", subStr);

            else if (validIdentifier(subStr) == false
                    && isDelimiter(str[right - 1]) == false)
                printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);
            left = right;
        }
    }
    return;
}

// DRIVER FUNCTION
int main()
{
    // maximum length of string is 100 here
    char str[100] = "int a = b + 1c; ";

    parse(str); // calling the parse function

    return (0);
}
```

2. 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;
}
```

3. Lex Program to recognize the strings which are starting and ending with 'a'

```
%{
#include<stdio.h>
%}
%%
(a|A)[a-z]*[0-9]*(a|A)   {printf("matching");}
(a|A)+   {printf("matching");}
.*   {printf("not matching");}
%%
main()
{
yylex();
return 0;
}
int yywrap()
{
}
```

Sample output

anna
matching
asssdf
not matching

4. Lex program to recognize Keywords.

```
%{
#include <stdio.h>;
%}
%%
if|else|while|int|switch|for|char   {printf("keyword");}
[a-z]([a-z]|[0-9])*   {printf("identifier");}
[0-9]*      {printf("number");}
.*   {printf("invalid");}
%%
main()
{
yylex();
return 0;
}
int yywrap()
{
}
```

Sample output

else
keyword
humble
identifier
9876
number

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

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

6. Lex program to recognize Identifiers.

4 lo chusko pooooo

7. Lex program to assign line numbers for source code.

```
/* Program to add line numbers
to a given file*/
%{
int line_number = 1; // initializing line number to 1
```

```
%}

/* simple name definitions to simplify the scanner specification name definition of line*/

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

/* whenever a line is encountered increment count*/

/* 10 specifies the padding from left side to present the line numbers*/

/* yytext The text of the matched pattern is stored in this variable (char*)*/

%%

int yywrap(){}

int main(int argc, char*argv[])
{
extern FILE *yyin; // yyin as pointer of File type

yyin = fopen("testtest.c","r"); /* yyin points to the file testtest.c and opens it in read mode.*/

yylex(); // The function that starts the analysis.

return 0;
}
```

8. Implement lexical analyzer in 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(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}
```

9. Write a program to find first and follow set of the variable in the given productions.

```
// C program to calculate the First and
// Follow sets of a given grammar
#include<stdio.h>
#include<ctype.h>
#include<string.h>

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;
```

```
// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
	int jm = 0;
	int km = 0;
	int i, choice;
	char c, ch;
	count = 8;

	// The Input grammar
	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;

	// Initializing the calc_first array
	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;
```

```
        // Checking if First of c has
        // already been calculated
        for(kay = 0; kay <= ptr; kay++)
            if(c == done[kay])
                xxx = 1;

        if (xxx == 1)
            continue;

        // Function call
        findfirst(c, 0, 0);
        ptr += 1;

        // Adding c to the calculated list
        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        // Printing the First Sets of the grammar
        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");
    printf("-----------------------------------------------\n\n");
    char donee[count];
    ptr = -1;

    // Initializing the calc_follow array
    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;

                // Checking if Follow of ck
                // has already been calculated
                for(kay = 0; kay <= ptr; kay++)
                        if(ck == donee[kay])
                                xxx = 1;

                if (xxx == 1)
                        continue;
                land += 1;

                // Function call
                follow(ck);
                ptr += 1;

                // Adding ck to the calculated list
                donee[ptr] = ck;
                printf(" Follow(%c) = { ", ck);
                calc_follow[point1][point2++] = ck;

                // Printing the Follow Sets of the grammar
                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;

	// Adding "$" to the follow
	// set of the start symbol
	if(production[0][0] == c) {
		f[m++] = '$';
	}
	for(i = 0; i < 10; i++)
	{
		for(j = 2;j < 10; j++)
		{
			if(production[i][j] == c)
			{
				if(production[i][j+1] != '\0')
				{
					// Calculate the first of the next
					// Non-Terminal in the production
					followfirst(production[i][j+1], i, (j+2));
				}

				if(production[i][j+1]=='\0' && c!=production[i][0])
				{
					// Calculate the follow of the Non-Terminal
					// in the L.H.S. of the production
					follow(production[i][0]);
				}
			}
		}
	}
}

void findfirst(char c, int q1, int q2)
{
	int j;

	// The case where we
	// encounter a Terminal
	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))
                        {
                            // Recursion to calculate First of New
                            // Non-Terminal we encounter after epsilon
                            findfirst(production[q1][q2], q1, (q2+1));
                        }
                        else
                            first[n++] = '#';
                    }
                    else if(!isupper(production[j][2]))
                    {
                        first[n++] = production[j][2];
                    }
                    else
                    {
                        // Recursion to calculate First of
                        // New Non-Terminal we encounter
                        // at the beginning
                        findfirst(production[j][2], j, 3);
                    }
                }
        }
}

void followfirst(char c, int c1, int c2)
{
        int k;

        // The case where we encounter
        // a Terminal
        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;
                }

                //Including the First set of the
                // Non-Terminal in the Follow of
                // the original query
                while(calc_first[i][j] != '!')
                {
                        if(calc_first[i][j] != '#')
                        {
                                f[m++] = calc_first[i][j];
                        }
                        else
```

```
			{
				if(production[c1][c2] == '\0')
				{
					// Case where we reach the
					// end of a production
					follow(production[c1][0]);
				}
				else
				{
					// Recursion to the next symbol
					// in case we encounter a "#"
					followfirst(production[c1][c2], c1, c2+1);
				}
			}
			j++;
		}}}
```

10. Write a program to find follow set of the variable in the given productions.

Above…………

11. Write a program for Recursive descent Parsing for expression grammar.

```
#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;
}
else
return 0;
}
int Tdash()
{
if(*ip=='*')
{
```

```
printf("%s\tT'->*FT' \n",ip);
ip++;
if(F())
{
if(Tdash())
{
return 1;
}
else
return 0;
}
else
return 0;
}
else
{
printf("%s\tT'->^ \n",ip);
return 1;
}
}
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;
}
```

12. Implement LL(1) Parser.

```
#include<stdio.h>
#include<string.h>
#define TSIZE 128
// table[i][j] stores the index of production that must be applied on ith
varible if the input is jth nonterminal
int table[100][TSIZE];
// stores all list of terminals the ASCII value if use to index terminals
terminal[i] = 1 means the character with ASCII value is a terminal
char terminal[TSIZE];
// stores all list of terminals only Upper case letters from 'A' to 'Z'
can be nonterminals nonterminal[i] means ith alphabet is present as
nonterminal is the grammar
char nonterminal[26];
//structure to hold each production str[] stores the production len is the
length of production
struct product {
char str[100];
int len;
}pro[20];
// no of productions in form A->ß
int no_pro;
char first[26][TSIZE];
char follow[26][TSIZE];
// stores first of each production in form A->ß
char first_rhs[100][TSIZE];
// check if the symbol is nonterminal
int isNT(char c) {
return c >= 'A' && c <= 'Z';
}
// reading data from the file
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];
}
}
// Calculates FIRST(ß) for each A->ß
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();
FIRST_RHS();
int i, j, k;
// display first of each variable
printf("\n");
for (i = 0; i < no_pro; ++i) {
if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
char c = pro[i].str[0];
printf("FIRST OF %c: ", c);
for (j = 0; j < TSIZE; ++j) {
if (first[c - 'A'][j]) {
printf("%c ", j);
}
}
printf("\n");
}
}
// display follow of each variable
printf("\n");
for (i = 0; i < no_pro; ++i) {
if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
char c = pro[i].str[0];
printf("FOLLOW OF %c: ", c);
for (j = 0; j < TSIZE; ++j) {
if (follow[c - 'A'][j]) {
printf("%c ", j);
}
}
```

```
printf("\n");
}
}
// display first of each variable ß
// in form A->ß
printf("\n");
for (i = 0; i < no_pro; ++i) {
printf("FIRST OF %s: ", pro[i].str);
for (j = 0; j < TSIZE; ++j) {
if (first_rhs[i][j]) {
printf("%c ", j);
}
}
printf("\n");
}
// the parse table contains '$'
// set terminal['$'] = 1
// to include '$' in the parse table
terminal['$'] = 1;
// the parse table do not read '^'
// as input
// so we set terminal['^'] = 0
// to remove '^' from terminals
terminal['^'] = 0;
// printing parse table
printf("\n");
printf("\n\t**************** LL(1) PARSING TABLE *******************\n");
printf("\t--------------------------------------------------------\n");
printf("%-10s", "");
for (i = 0; i < TSIZE; ++i) {
if (terminal[i]) printf("%-10c", i);
}
printf("\n");
int p = 0;
for (i = 0; i < no_pro; ++i) {
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");
}
}
}
```

Lab Program

Write a program to generate predictive LL1 parsing table for the given grammar

```
from collections import defaultdict
import pandas as pd
First = defaultdict(list)
Follow = defaultdict(list)
def first(productionSet,nt,parent):
    for x in productionSet[nt[0]]:
        if not x[0].isupper():
            if x[0] not in First[parent]:
                First[parent].append(x[0])
        else:
            first(productionSet,x[0],parent)

def follow(productionSet,nt,parent):
    d=defaultdict(list)
    for key,val in productionSet.items():
        for x in val:
            if nt in x and len(x)!= 1:
                d[key].append(x)
    Keys = list(productionSet.keys())
    if nt == Keys[0]:
        Follow[nt] = ['$']
    elif len(d)==0:
```

```
        return
    for key,val in d.items():
        for x in val:
            if x[-1] == nt:
                alpha,B,beta = x[:-1],nt,None
            elif x[0] == nt:
                alpha,B,beta = None,nt,x[1:]
            else:
                idx_B = x.index(nt)
                alpha,B,beta = x[:idx_B],x[idx_B],x[idx_B+1:]
            if beta == None or beta == 'e':
                if key in Follow:
                    Follow[nt].extend(Follow[key])
                else:
                    if key != nt:
                        follow(productionSet,key,parent)
                        if key in Follow:
                            Follow[nt].extend(Follow[key])


            elif beta[0] not in First:
                Follow[nt].extend([beta[0]])
            else:
                temp =[]
                for v in First[beta[0]]:
                    if v == 'e':
                        if key in Follow:
```

```
                temp.extend(Follow[key])
            else:
                if key != nt:
                    follow(productionSet,key,parent)
                    if key in Follow:
                        Follow[nt].extend(Follow[key])
        else:
            temp.append(v)
    Follow[nt].extend(temp)


def predictiveParser(productionSet):
    table = defaultdict(dict)
    terminals = ['$']
    for key, val in productionSet.items():
        for prod in val:
            for x in prod:
                if not x.isupper() and x not in terminals:
                    terminals.append(x)
    terminals.sort()
    nonTerminals = list(First.keys())
    for x in terminals:
        First[x] = [x]
    for nt in nonTerminals:
        for t in terminals:
            table[nt][t] = []
    for key,val in productionSet.items():
```

```
        for v in val:
            f = First[v[0]][:]
            if 'e' in f:
                f.remove('e')
                f.extend(Follow[key])
            for x in f:
                if key+'->'+v not in table[key][x]:
                    table[key][x].append(key+'->'+v)
    t = pd.DataFrame.from_dict(table,orient='index')
    pd.set_option('display.max_rows', None)
    pd.set_option('display.max_columns', None)
    print(t)
# productionSet = {'E' : ['TZ'],'Z' : ['+TZ','e'],'T':['FY'],'Y': ['*FY','e'],'F':['(E)','i']}
# productionSet = {'S': ['ACB', 'CbB', 'Ba'], 'A': ['da', 'BC'], 'B': ['g', 'e'], 'C': ['h', 'e']}
productionSet = {'S': ['aBD'], 'B': ['cC'], 'C': ['bc', 'e'], 'D': ['EF'],'E':['g','e'],'F':['f','e']}
print(productionSet)
for key,val in productionSet.items():
    first(productionSet,key,key)

for key,val in First.items():
    print("First({0}) = {1}".format(key,val))
for key,val in productionSet.items():
    follow(productionSet,key,key)
for key,val in Follow.items():
    print("Follow({0}) = {1}".format(key,set(val)))
predictiveParser(productionSet)
```

Output:

Testcase #1

```
{'S': ['aBD'], 'B': ['cC'], 'C': ['bc', 'e'], 'D': ['EF'], 'E': ['g', 'e'], 'F': ['f', 'e']}
First(S) = ['a']
First(B) = ['c']
First(C) = ['b', 'e']
First(D) = ['g', 'e']
First(E) = ['g', 'e']
First(F) = ['f', 'e']
Follow(S) = {'$'}
Follow(B) = {'g', '$'}
Follow(C) = {'g', '$'}
Follow(D) = {'$'}
Follow(E) = {'f', '$'}
Follow(F) = {'$'}
          $         a         b         c   e       f         g
S        []   [S->aBD]        []        []  []      []        []
B        []         []        []   [B->cC]  []      []        []
C    [C->e]         []   [C->bc]        []  []      []    [C->e]
D   [D->EF]         []        []        []  []      []   [D->EF]
E    [E->e]         []        []        []  []  [E->e]    [E->g]
F    [F->e]         []        []        []  []  [F->f]        []


...Program finished with exit code 0
Press ENTER to exit console.
```

Testcase #2

```
{'E': ['TZ'], 'Z': ['+TZ', 'e'], 'T': ['FY'], 'Y': ['*FY', 'e'], 'F': ['(E)', 'i']}
First(E) = ['(', 'i']
First(Z) = ['+', 'e']
First(T) = ['(', 'i']
First(Y) = ['*', 'e']
First(F) = ['(', 'i']
Follow(E) = {'$', ')'}
Follow(Z) = {'$', ')'}
Follow(T) = {'$', '+', ')'}
Follow(Y) = {'$', '+', ')'}
Follow(F) = {'*', '$', '+', ')'}
          $          (         )          *          +   e         i
E        []    [E->TZ]        []         []         []  []   [E->TZ]
Z    [Z->e]         []    [Z->e]         []   [Z->+TZ]  []        []
T        []    [T->FY]        []         []         []  []   [T->FY]
Y    [Y->e]         []    [Y->e]   [Y->*FY]     [Y->e]  []        []
F        []   [F->(E)]        []         []         []  []    [F->i]


...Program finished with exit code 0
Press ENTER to exit console.
```

LAB PROGRAMS

Implement SLR parser.

```
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])
          return newRules
def findClosure(input_state, dotSymbol):
          global start_symbol, \
                    separatedRulesList, \
                    statesDict
          closureSet = []
          if dotSymbol == start_symbol:
                    for rule in separatedRulesList:
                              if rule[0] == dotSymbol:
                                        closureSet.append(rule)
          else:
                    closureSet = input_state
          prevLen = -1
          while prevLen != len(closureSet):
                    prevLen = len(closureSet)
                    tempClosureSet = []
                    for rule in closureSet:
                              indexOfDot = rule[1].index('.')
                              if rule[1][-1] != '.':
                                        dotPointsHere = rule[1][indexOfDot + 1]
                                        for in_rule in separatedRulesList:
                                                  if dotPointsHere == in_rule[0] and \
                                                                      in_rule not in tempClosureSet:
                                                            tempClosureSet.append(in_rule)
                    for rule in tempClosureSet:
                              if rule not in closureSet:
                                        closureSet.append(rule)
          return closureSet
def compute_GOTO(state):
          global statesDict, stateCount
          generateStatesFor = []
          for rule in statesDict[state]:
```

```
        if rule[1][-1] != '.':
            indexOfDot = rule[1].index('.')
            dotPointsHere = rule[1][indexOfDot + 1]
            if dotPointsHere not in generateStatesFor:
                generateStatesFor.append(dotPointsHere)
    if len(generateStatesFor) != 0:
        for symbol in generateStatesFor:
            GOTO(state, symbol)
    return
def GOTO(state, charNextToDot):
    global statesDict, stateCount, stateMap
    newState = []
    for rule in statesDict[state]:
        indexOfDot = rule[1].index('.')
        if rule[1][-1] != '.':
            if rule[1][indexOfDot + 1] == \
                    charNextToDot:
                shiftedRule = copy.deepcopy(rule)
                shiftedRule[1][indexOfDot] = \
                    shiftedRule[1][indexOfDot + 1]
                shiftedRule[1][indexOfDot + 1] = '.'
                newState.append(shiftedRule)
    addClosureRules = []
    for rule in newState:
        indexDot = rule[1].index('.')
        if rule[1][-1] != '.':
            closureRes = \
                findClosure(newState, rule[1][indexDot + 1])
            for rule in closureRes:
                if rule not in addClosureRules \
                        and rule not in newState:
                    addClosureRules.append(rule)
    for rule in addClosureRules:
        newState.append(rule)
    stateExists = -1
    for state_num in statesDict:
        if statesDict[state_num] == newState:
            stateExists = state_num
            break
    if stateExists == -1:
        stateCount += 1
        statesDict[stateCount] = newState
        stateMap[(state, charNextToDot)] = stateCount
    else:
        stateMap[(state, charNextToDot)] = stateExists
    return
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
				fres.append('#')
				return fres
def follow(nt):
	global start_symbol, rules, nonterm_userdef, \
		term_userdef, diction, firsts, follows
	solset = set()
	if nt == start_symbol:
		solset.add('$')
	for curNT in diction:
		rhs = diction[curNT]
		for subrule in rhs:
```

```
                        if nt in subrule:
                                while nt in subrule:
                                        index_nt = subrule.index(nt)
                                        subrule = subrule[index_nt + 1:]
                                        if len(subrule) != 0:
                                                res = first(subrule)
                                                if '#' in res:
                                                        newList = []
                                                        res.remove('#')
                                                        ansNew = follow(curNT)
                                                        if ansNew != None:
                                                                if type(ansNew) is list:
                                                                        newList = res + ansNew
                                                                else:
                                                                        newList = res + [ansNew]
                                                        else:
                                                                newList = res
                                                        res = newList
                                        else:
                                                if nt != curNT:
                                                        res = follow(curNT)
                                        if res is not None:
                                                if type(res) is list:
                                                        for g in res:
                                                                solset.add(g)
                                                else:
                                                        solset.add(res)
        return list(solset)
def createParseTable(statesDict, stateMap, T, NT):
        global separatedRulesList, diction
        rows = list(statesDict.keys())
        cols = T+['$']+NT
        Table = []
        tempRow = []
        for y in range(len(cols)):
                tempRow.append('')
        for x in range(len(rows)):
                Table.append(copy.deepcopy(tempRow))
        for entry in stateMap:
                state = entry[0]
                symbol = entry[1]
                # get index
                a = rows.index(state)
                b = cols.index(symbol)
                if symbol in NT:
                        Table[a][b] = Table[a][b]\
                                + f"{stateMap[entry]} "
                elif symbol in T:
                        Table[a][b] = Table[a][b]\
                                + f"S{stateMap[entry]} "
        numbered = {}
```

```
	key_count = 0
	for rule in separatedRulesList:
		tempRule = copy.deepcopy(rule)
		tempRule[1].remove('.')
		numbered[key_count] = tempRule
		key_count += 1
	addedR = f"{separatedRulesList[0][0]} -> " \
		f"{separatedRulesList[0][1][1]}"
	rules.insert(0, addedR)
	for rule in rules:
		k = rule.split("->")
		k[0] = k[0].strip()
		k[1] = k[1].strip()
		rhs = k[1]
		multirhs = rhs.split('|')
		for i in range(len(multirhs)):
			multirhs[i] = multirhs[i].strip()
			multirhs[i] = multirhs[i].split()
		diction[k[0]] = multirhs
	for stateno in statesDict:
		for rule in statesDict[stateno]:
			if rule[1][-1] == '.':
				temp2 = copy.deepcopy(rule)
				temp2[1].remove('.')
				for key in numbered:
					if numbered[key] == temp2:
						follow_result = follow(rule[0])
						for col in follow_result:
							index = cols.index(col)
							if key == 0:
								Table[stateno][index] = "Accept"
							else:
								Table[stateno][index] =\
									Table[stateno][index]+f"R{key} "
	print("\nSLR(1) parsing table:\n")
	frmt = "{:>8}" * len(cols)
	print(" ", frmt.format(*cols), "\n")
	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")
separatedRulesList = \
          grammarAugmentation(rules,
                                                  nonterm_userdef,
                                                  start_symbol)
printResult(separatedRulesList)
start_symbol = separatedRulesList[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)
```

OUTPUT:

```
Original grammar input:

E -> E + T | T
T -> T * F | F
F -> ( E ) | id

Grammar after Augmentation:

E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

Calculated closure: I0

E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

States Generated:

State = I0
E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id
```

```
State = I2
E -> T .
T -> T . * F

State = I3
T -> F .

State = I4
F -> ( . E )
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I5
F -> id .

State = I6
E -> E + . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I7
T -> T * . F
F -> . ( E )
F -> . id

State = I8
F -> ( E . )
E -> E . + T

State = I9
```

```
State = I9
E -> E + T .
T -> T . * F

State = I10
T -> T * F .

State = I11
F -> ( E ) .

Result of GOTO computation:

GOTO ( I0 , E ) = I1
GOTO ( I0 , T ) = I2
GOTO ( I0 , F ) = I3
GOTO ( I0 , ( ) = I4
GOTO ( I0 , id ) = I5
GOTO ( I1 , + ) = I6
GOTO ( I2 , * ) = I7
GOTO ( I4 , E ) = I8
GOTO ( I4 , T ) = I2
GOTO ( I4 , F ) = I3
GOTO ( I4 , ( ) = I4
GOTO ( I4 , id ) = I5
GOTO ( I6 , T ) = I9
GOTO ( I6 , F ) = I3
GOTO ( I6 , ( ) = I4
GOTO ( I6 , id ) = I5
GOTO ( I7 , F ) = I10
GOTO ( I7 , ( ) = I4
GOTO ( I7 , id ) = I5
GOTO ( I8 , ) ) = I11
GOTO ( I8 , + ) = I6
GOTO ( I9 , * ) = I7
```

SLR(1) parsing table:

	id	+	*	(	)	$	E	T	F
I0	S5			S4			1	2	3
I1		S6				Accept			
I2		R2	S7		R2	R2			
I3		R4	R4		R4	R4			
I4	S5			S4			8	2	3
I5		R6	R6		R6	R6			
I6	S5			S4				9	3
I7	S5			S4					10
I8		S6			S11				
I9		R1	S7		R1	R1			
I10		R3	R3		R3	R3			
I11		R5	R5		R5	R5			

Implement parser generator using YACC(calculator)

Lex program:

```
%{
#include &quot;y.tab.h&quot;
#include&lt;math.h&gt;
%}
%%
([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&lt;stdio.h&gt;
#include&lt;math.h&gt;
double memvar;
%}
%union
{

double dval;
}
%token&lt;dval&gt;NUMBER
%token&lt;dval&gt;MEM
%token LOG SINE nLOG COS TAN
%left &#39;-&#39;&#39;+&#39;;
%left &#39;*&#39;&#39;/&#39;;
%right &#39;^&#39;;
%left LOG SINE nLOG COS TAN
%nonassoc UMINUS
%type&lt;dval&gt; expression
%%
start: statement &#39;\n&#39;;
|start statement &#39;\n&#39;;
;
statement: MEM&#39;=&#39;expression { memvar=$3;}
|expression {printf(&quot;answer=%g\n&quot;,$1);}
;
expression:expression&#39;+&#39;expression {$$=$1+$3;}
|expression&#39;-&#39;expression {$$=$1+$3;}
```

```
|expression&#39;*&#39;expression {$$=$1*$3;}
|expression&#39;/&#39;expression
{
if($3==0)
yyerror(&quot;divide by zero&quot;);
else
$$=$1/$3;}
|expression&#39;^&#39;expression {$$=pow($1,$3);}
;
expression: &#39;-&#39; expression %prec UMINUS {$$=-$2;}
|&#39;(&#39;expression&#39;)&#39; {$$=$2;}
|LOG expression {$$=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(&quot;enter expression:&quot;);
yyparse();
}
int yyerror(char *error)

{
fprintf(stderr,&quot;%s\n&quot;,error);
}
yywrap()
{ return 1;
}
```

OUTPUT:

```
[cse19080@CCLINUXSERVER ~]$ lex calculator.l
[cse19080@CCLINUXSERVER ~]$ yacc -d calculator.y
[cse19080@CCLINUXSERVER ~]$ gcc lex.yy.c y.tab.c -ll -lm
[cse19080@CCLINUXSERVER ~]$ ./a.out
enter expression: 2*3+5
answer=11
2.5*4
answer=10
2.5e3+1
answer=2501
```

Lab Program

C program for Three address code generation.

Code:

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;
}
```

Output:

```
enter expression (a*b)+(c*d)
        three address codes

B:=   a    *    b
C:=   c    *    d
D:=   B    +    C
```

WEEK-7
LAB PROGRAMS

Implement CLR parser.

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) #union operation
    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]: #this is okay since I'm using an OrderedDict
        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)
#(Format: "A->Y1Y2..Yn" {Yi - single char} OR "A->" {epsilon})''')
  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):
        if chr(i) not in nt_list:
            start_prod=production_list[0]
            production_list.insert(0, chr(i)+'->'+start_prod.split('->')[0])
            return
def main():
    global production_list, ntl, nt_list, tl, t_list
    firstfollow.main()
    print("\tFIRST AND FOLLOW OF NON-TERMINALS")
    for nt in ntl:
        firstfollow.compute_first(nt)
        firstfollow.compute_follow(nt)
        print(nt)
        print("\tFirst:\t", firstfollow.get_first(nt))
        print("\tFollow:\t", firstfollow.get_follow(nt), "\n")
    augment_grammar()
    nt_list=list(ntl.keys())
    t_list=list(tl.keys()) + ['$']
    print(nt_list)
    print(t_list)
    j=calc_states()
    ctr=0
    for s in j:
        print("Item{}:".format(ctr))
        for i in s:
            print("\t", i)
        ctr+=1
    table=make_table(j)
    print('_____________________________________________________________________')
    print("\n\tCLR(1) TABLE\n")
    sym_list = nt_list + t_list
    sr, rr=0, 0
    print('_____________________________________________________________________')
    print('\t|  ','\t|  '.join(sym_list),'\t\t|')
    print('_____________________________________________________________________')
    for i, j in table.items():
        print(i, "\t|  ", '\t|  '.join(list(j.get(sym,' ') if type(j.get(sym))in (str , None) else next(iter(j.get(sym,' ')))  for sym in 
sym_list)),'\t\t|')
        s, r=0, 0
        for p in j.values():
            if p!='accept' and len(p)>1:
                p=list(p)
                if('r' in p[0]): r+=1
                else: s+=1
                if('r' in p[1]): r+=1
                else: s+=1
        if r>0 and s>0: sr+=1
        elif r>0: rr+=1
    print('_____________________________________________________________________')
```

```
    print("\n", sr, "s/r conflicts |", rr, "r/r conflicts")
    print('______________________________________________________________________')
    print("Enter the string to be parsed")
    Input=input()+'$'
    try:
        stack=['0']
        a=list(table.items())
        '''print(a[int(stack[-1])][1][Input[0]])
        b=list(a[int(stack[-1])][1][Input[0]])
        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()
```

OUTPUT:

```
Enter the grammar productions (enter 'end' or return to stop)
#(Format: "A->Y1Y2..Yn" {Yi - single char} OR "A->" {epsilon})
S->AA
A->aA
A->b
end
        FIRST AND FOLLOW OF NON-TERMINALS
S
        First:  {'a', 'b'}
        Follow: {'$'}

A
        First:  {'a', 'b'}
        Follow: {'a', 'b', '$'}

['S', 'A']
['a', 'b', '$']
Item0:
         Z->.S, $
         S->.AA, $
         A->.aA, a|b
         A->.b, a|b
Item1:
         Z->S., $
Item2:
         S->A.A, $
         A->.aA, $
         A->.b, $
Item3:
         A->a.A, a|b
         A->.aA, a|b
         A->.b, a|b
Item4:
         A->b., a|b
Item5:
         S->AA., $
Item6:
        A->a.A, $
        A->.aA, $
        A->.b, $
Item7:
        A->b., $
Item8:
        A->aA., a|b
Item9:
        A->aA., $
_______________________________________________________________

        CLR(1) TABLE

_______________________________________________________________
        |  S   |  A   |  a   |  b   |  $              |
_______________________________________________________________
0       |  1   |  2   |  s3  |  s4  |                 |
1       |      |      |      |      |  accept         |       |
2       |      |  5   |  s6  |  s7  |                 |
3       |      |  8   |  s3  |  s4  |                 |
4       |      |      |  r3  |  r3  |                 |
5       |      |      |      |      |  r1             |
6       |      |  9   |  s6  |  s7  |                 |
7       |      |      |      |      |  r3             |
8       |      |      |  r2  |  r2  |                 |
9       |      |      |      |      |  r2             |
_______________________________________________________________

 0 s/r conflicts | 0 r/r conflicts
_______________________________________________________________
Enter the string to be parsed
aaabab
productions     : ['Z->S', 'S->AA', 'A->aA', 'A->b']
stack                          Input
0                             aaabab$
0a3                           aabab$
0a3a3                         abab$
0a3a3a3                       bab$
0a3a3a3b4                             ab$
0a3a3a3A8                             ab$
0a3a3A8                       ab$
0a3A8                         ab$
0A2                           ab$
0A2a6                         b$
0A2a6b7                       $
0A2a6A9                       $
0A2A5                         $
0S1                           $
```

Construct DAG for 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);
    }
}
```

Input / Output:

```
$ gcc dag.c
$ ./a.out
Enter the postfix expression:
abcd+*-
a       abcd+*-
The tree is created.....
Inorder DAG is :
a-b*c+d
```

WEEK-8
LAB PROGRAM

C Program to implement a code optimization method “common sub expression elimination”.

```
#include<stdio.h>
#include<string.h>
int tc[10],fb=0,i=0,j=0,k=0,p=0,fstar=0,c=-1,c1=0,c2=0,t1,t2,t3,t4,fo=0;
char m[30],temp[30],opt[10][4];
operatormajid(char haj,char haj1)
{
  m1: for(i=0;m[i]!='\0';i++)
  if(m[i]==haj||m[i]==haj1)
  {
    fstar++;
    break;
  }
  if(fstar==1)
  {
    for(j=0;j<i;j++)
    if(m[j]=='T')c++;
    printf("\nT%d=",k);
    if(m[i-1]=='T'&&m[i+1]=='T')
    {
      printf("%c%d%c%c%d",m[i-1],tc[c],m[i],m[i+1],tc[c+1]);
      tc[c]=k++;
      for(t2=c+1;t2<9;t2++)
        tc[t2]=tc[t2+1];
    }
    else if(m[i-1]!='T'&&m[i+1]!='T')
    {
      printf("%c%c%c",m[i-1],m[i],m[i+1]);
      if(c==-1)
      {
        for(t1=9;t1>0;t1--)
          tc[t1]=tc[t1-1];
        tc[0]=k++;
      }
      else if(c>=0)
      {
        for(t1=9;t1>c+1;t1--)
          tc[t1]=tc[t1-1];
        tc[t1]=k++;
      }
    }
    else if(m[i-1]=='T'&&m[i+1]!='T')
    {
      printf("%c%d%c%c",m[i-1],tc[c],m[i],m[i+1]);
      tc[c]=k++;
```

```
    }
    else if(m[i-1]!='T'&&m[i+1]=='T')
    {
      printf("%c%c%c%d",m[i-1],m[i],m[i+1],tc[c+1]);
      tc[c+1]=k++;
    }
    for(t1=0;t1<i-1;t1++)
      temp[t1]=m[t1];
    temp[t1++]='T';
    for(t2=i+2;m[t2]!='\0';t2++)
      temp[t1++]=m[t2];
    temp[t1++]='\0';
    fstar=0;
    for(i=0;temp[i]!='\0';i++)
      m[i]=temp[i];
    m[i]='\0';
    c=-1;
    goto m1;
  }
  else
    return 0;
}
int main()
{
  int a,d;
  for(i=0;i<10;i++)
  tc[i]=-1;
  printf("\n Code stmt evaluation follow following precedence: ");
  printf("\n 1.( ) within the () stmt should be of the form: x op z");
  printf("\n 2.*,/ equal precedence");
  printf("\n 3.+,- equal precedence");
  printf("\n Enter ur Code Stmt-");
  gets(m);
  i=0;
  while(m[i]!='\0'){
    if(m[i++]=='('){
      fb++;
      break;
    }
  }
  i=0;
  printf("\nThe Intermediate Code may generated as-");
  if(fb==1)
  { /* evaluating sub exp */
    while(m[i]!='\0')
      if(m[i]=='(')
      {
        temp[j++]='T';
```

```
          i++;
          t3=i; /* optimising the code */
          while(m[i]!=')')
            opt[c1][c2++]=m[i++];
          for(t4=c1-1;t4>=0;t4--)
            if(strcmp(opt[c1],opt[t4])==0)
            {
              tc[p++]=t4;
              fo=1;
            } /* end of optimising */
          if(fo==0)
          {
            tc[p++]=k++;
            printf("\nT%d=",k-1);
            while(m[t3]!=')')
              printf("%c",m[t3++]);
          }
          i++;
          c1++;
          c2=fo=0;
        }
        else if(m[i]!='(')
          temp[j++]=m[i++];
        if(fb==1)
        {
          temp[j]='\0';
          for(i=0;temp[i]!='\0';i++)
            m[i]=temp[i];
            m[i]='\0';
        }
   } /* end of evluating sub exp */
   a=operatormajid('*','/'); /* operator fun call depends on priority */
   d=operatormajid('+','-');
   if(a==0&&d==0&&m[1]=='=')
     printf("\n%s%d",m,k-1);
   getch();
}
```

OUTPUT:

```
 Code stmt evaluation follow following precedence:
 1.( ) within the () stmt should be of the form: x op z
 2.*,/ equal precedence
 3.+,- equal precedence
 Enter ur Code Stmt-a+(b*c)-d/(b*c)

The Intermediate Code may generated as-
T0=b*c
T1=d/T0
T2=a+T0
T3=T2-T1
```

WEEK-9
LAB PROGRAM

Program to generate machine code.

```
#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){
```

```
//printf("in check_label");
int i;for(i=0;i<no;i++){
if(k==label[i])
return 1;
}
return 0;
}
```

OUTPUT:

```
enter filename of intermediate code:target.c


        STORE t1, 2

        STORE a[0],1

        STORE a[1],2

        STORE a[2],3

        LOAD t1,R0
        LOAD 6,R1
        MUL R1,R0
        STORE R0,t2

        LOAD a[2],R0
        LOAD t2,R1
        ADD R1,R0
        STORE R0,t3

        LOAD a[t2],R0
        LOAD t1,R1
        SUB R1,R0
        STORE R0,t2

        LOAD 3,R0
        LOAD t2,R!
        DIV R1,R0
        STORE R0,t2

        LOAD t2, R1
        STORE R1,t2

        LOAD t2,R0
        JGT 5,label.# 11

label.# 11:
        JMP t2 label.# 13

        STORE t3, 99

label.# 13:
        LOAD t3,R0
        LOAD t4,R1
        ADD R1,R0
        STORE R0,printf
```

```
= t1 2
[]= a 0 1
[]= a 1 2
[]= a 2 3
* t1 6 t2
+ a[2] t2 t3
- a[t2] t1 t2
/ 3 t2 t2
uminus t2 t2
> t2 5 11
goto t2 13
= t3 99
+ t3 t4
printf t4
```