**1**A. Program For converting all small case letters to UPPER case letters and Vice-Versa.

**1** B. Program to count the words, spaces, and lines in a given input file.

2A. LEX program to eliminate comment lines (Single and Multiple) in a text(C program)  file and copy the resulting program into a separate file.

2B. YACC program to recognize valid identifier, operators and keywords in the given text  (C program) file.

3A Develop a LEX program to recognize valid arithmetic expression. Identifiers in the expression

could be only integers and operators could be + and \*. Count the identifiers & operators

present and print them separately.

3B. Develop a YACC program to evaluate arithmetic expression involving operators: +, -, \*, and /.

4  Develop, Implement and execute a program using YACC/Bison tool to recognize all strings

ending with b preceded by n a’s using the grammar a n b (note: input n value), also create DFA

of given grammar using JFLAP

5 Develop a program to find FIRST and FOLLOW of all variables. Write a suitable data structure to

store a context fee grammar. Prerequisite is to eliminate left recursion from the grammar

before storing.

1 a)

Flex prac1.l

Gcc lex.yy.c

a.exe

%{

#include <stdio.h>

%}

%%

[a-z] { printf("%c", 'A' + yytext[0] - 'a'); }

[A-Z] { printf("%c", 'a' + yytext[0] - 'A'); }

. { printf("%c", yytext[0]); }

%%

int main(void) {

 yylex();

 yywrap();

 return 0;

}

int yywrap(){

return 1;}

1 b

%{

#include <stdio.h>

int word\_count = 0;

int space\_count = 0;

int line\_count = 0;

int number\_count = 0;

%}

%%

[a-zA-Z]+ { word\_count++; }

[ \t] { space\_count++; }

\n { ++line\_count; }

[0-9]\* { number\_count++; }

. { /\* ignore other characters \*/ }

%%

int main() {

FILE \*file = fopen("input.txt", "r");

if (file == NULL) {

perror("Error opening file");

return 1;

}

yyin = file;

yylex();

yywrap();

fclose(file);

printf("Words: %d\nSpaces: %d\nLines: %d\nnumbers :%d\n",

word\_count, space\_count, line\_count , number\_count);

return 0;

}

int yywrap()

{

return 1;

}

//make the input file in the this question

**2.a**

%{

#include<stdio.h>

int sl=0;

int ml=0;

%}

%%

"/\*"[a-zA-Z0-9' '\t\n]+"\*/" ml++;

"//".\* sl++;

%%

int yywrap()

{

return 1;

}

int main(){

yyin=fopen("in.txt","r");

yyout=fopen("ou.txt","w");

yylex();

fclose(yyin);

fclose(yyout);

return 0;

}

**2.b**

**.y file**

%{ int yywrap();

void yyerror();

int yyparse();

%}

%{

#include<stdio.h>

int id=0,key=0,op=0,valid=1;

extern FILE \*yyin;

%}

%token ID KEY OP

%%

input:ID {id++;}|KEY {key++;}|OP {op++;}|input ID {id++;}|input OP

{op++;}|input KEY {key++;}

%%

void yyerror()

{

valid=0;

}

int yywrap()

{

return 1;}

int main()

{

yyin=fopen("a.txt","r");

yyparse();

if(valid!=0)

printf("%d %d %d",id,op,key);

}

**.l file**

%{

#include<stdio.h>

#include "y.tab.h"

externintyylval;

%}

%%

[\t] ;

[+|-|\*|\|=] {printf("operator is %c\n",yytext[0]); return OP;}

int|main|if|printf|for|return|else {printf("\n keyword is %s\n",yytext); return KEY;}

[\_a-zA-Z][0-9A-Z]\* {printf("\n identifier is %s\n",yytext); return ID;}

[0-9][a-z][0-9a-z]\* ;

"(".\*");" ;

. ;

%%

**3.a .l file**

%{

#include<stdio.h>

int v=0, op=0, id=0;

%}

%%

[a-zA-Z]+[0-9A-Za-z]\* { id++; }

[0-9]+ { id++; }

[\+\-\\*/\=] { op++; }

"(" { v++; }

")" { v--; }

";" { /\* Do nothing, used as delimiter \*/ }

.|\n { /\* Ignore other characters \*/ }

%%

int yywrap() {

return 1;

}

int main() {

printf("Enter the expression:\n");

yylex();

if ((op + 1) == id && v == 0) {

printf("\nExpression is Valid\n");

printf("Number of identifiers = %d \nNumber of operators = %d \n", id, op);

} else {

printf("\nExpression is Invalid\n");

}

return 0;

}

**3.b**

**.l file**

%{

#include "y.tab.h" // This is required to include the token definitions

%}

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

[\t ]+ { /\* ignore whitespace \*/ }

\n { return NL; }

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

"(" { return '('; }

")" { return ')'; }

. { printf("Unexpected character: %s\n", yytext); exit(1); }

%%

int yywrap() {

return 1;

}

**.y file**

%{

#include<stdio.h>

#include<stdlib.h>

extern int yylex();

extern int yyerror(char \*);

int yyparse();

%}

%token NUMBER NL

%left '+' '-'

%left '\*' '/'

%%

stmt: exp NL {printf("VALUE= %d\n",$1); exit(0);}

;

exp: exp '+' exp {$$ = $1 + $3 ;} |

exp '-' exp {$$ = $1 - $3 ; } |

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

exp '/' exp { if ($3 == 0) {printf("cannot divide by 0\n"); exit(0);} else {$$ = $1 / $3 ;}} |

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

NUMBER { $$ = $1 ; }

;

%%

int yyerror(char \*msg)

{

printf("Invalid expression\n");

exit(0);

}

int main()

{

printf("enter the expression: \n");

yyparse();

return 0;

}

**4 code**

**.l file**

%{

#include "y.tab.h"

extern int yylval;

%}

A [a]

B [b]

%%

{A} {yylval=yytext[0];return A;}

{B} {yylval=yytext[1];return B;}

\n {return 0;}

. {return yytext[0];}

%%

int yywrap()

{

return 1;

}

**.y file**

%{

#include<stdio.h>

int yylex(void);

int yyerror(char \*);

%}

%token A B //tokens : the alphabets of language 'a' and 'b'

%%

//production rules for grammar

expr: s B

;

s : s A

| A

;

%%

int main()

{

printf("Enter the string \n");

yyparse();

printf("Valid string");

return 0;

}

int yyerror(char \*s)

{

printf("Invalid: Not a part of the language - a^n b \n");

}

**5 code**

#include <ctype.h>

#include <stdio.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;

printf("Enter the number of productions: ");

scanf("%d", &count);

printf("Enter the productions in the format 'Non-Terminal=Production':\n");

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

scanf("%s", production[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++;

}

}

}