ex-1 # Ex.No.1 - Develop a program in C to design a lexical analyzer that recognizes identifiers and constants

lex.c

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

void main() {
	char in[50], temp[50];
	int i = 0, j = 0;
	printf("Enter the expression : ");
	gets(in);
	printf("\nIdentifier \t Constant\n");

	while(in[i] != '\0') {
		if ( isalpha(in[i]) ) {

			while ( isalpha(in[i]) || isdigit(in[i]) )
				temp[j++] = in[i++];

			temp[j] = '\0';
			printf("%s\n", temp);

		}
		else if (isdigit(in[i])) {
			while (isdigit(in[i]))
				temp[j++] = in[i++];

			temp[j] = '\0';
			printf("\t\t\t%s\n", temp);
		}
		else if (in[i] == ' ')
			i++;
		else
			i++;

		j=0;
	}
}
$ cc lex.c -o lex.out
```

Execution :

1. Navigate to your compiled program directory.
2. Run `./lex.out` in terminal.

OUTPUT:

```
Enter the expression : i = 2 + 3

Identifier      Constant
i
                  2
                  3
```

ex-2 # Ex.No.2 - Implementation a symbol table that involves insertion, deletion, search and modify operations using C language

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


struct sym_tab {
   char symbol[20];
   char type[20];
   int length;
};


struct sym_tab s[10];
int n = 0;


int main() {
   int ch;

   void insert();
   void del();
   void disp();
   void search();
   void modify();

   do {
      printf("\n1. Insert\n2. Delete\n3. Display\n4. Search\n5. Modify\n6. Exit\n");
      printf("\nEnter the choice : ");
      scanf("%d", &ch);

      switch(ch) {
         case 1:
            insert();

            break;

         case 2:
```

del();# Ex.No.2 - Implementation a symbol table that involves insertion, deletion, search and modify operations using C language

```
            break;

         case 3:
            disp();

            break;

         case 4:
            search();

            break;

         case 5:
            modify();

            break;

         default:

            break;
      }

   } while (ch < 6);

}


void insert() {
   char name[20], data[20];
   int leng, i, k, length;
   printf("Enter new symbol, datatype, length : ");
   scanf("%s%s%d", name, data, &leng);

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

      if ( strcmp(name, s[i].symbol) == 0 ) {
         printf("Duplicate entry\n");

         return;
      }

   }

   strcpy(s[n].symbol, name);
   strcpy(s[n].type, data);
   s[n].length = leng;
   n++;
}
```

```
void del() {
   int i, k;
   char sym[20];
   printf("Enter the symbol to be deleted : ");
   scanf("%s", sym);

   if (n == 0) {
      printf("Empty table\n");

      return;
   }

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

      if ( strcmp(sym, s[i].symbol) == 0 ) {

         for (k = i; k < n-1; k++) {
            strcpy(s[k].symbol, s[k+1].symbol);
            s[k].length = s[k+1].length;
         }

         n--;
         printf("The symbol is deleted\n");
      }

   }

}


void modify() {
   char name[20], data[20], old[20];
   int len,i;

   if (n == 0) {
      printf("empty tables\n");
      return;
   }

   printf("Enter the symbol to be modified : ");
   scanf("%s",old);

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

      if ( strcmp(old, s[i].symbol) == 0 ) {
         printf("Symbol is found %s \t%s \t\t%d", s[i].symbol, s[i].type, s[i].length);
         printf("\nEnter new values for datatypes, length : ");
         scanf("%s%d", data, &len);
         strcpy(s[i].type, data);
         s[i].length = len;
         printf("Symbol entries modified\n");
```

```
            return;
        }

    }

}


void search() {
    int i;
    char name[20];
    if (n == 0) {
        printf("Empty table\n");

        return;
    }

    printf("Enter the symbol to be searched : ");
    scanf("%s",name);

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

        if ( strcmp(name, s[i].symbol) == 0) {
            printf("Symbol found\n%s \t%s\t\t%d",s[i].symbol,s[i].type,s[i].length);

            return;
        }

    }

}


void disp() {
    int i;

    if(n==0) {
        printf("Empty table\n");

        return;
    }

    printf("Symbol\tdatatype\tlength\n");

    for( i = 0; i < n; i++)
        printf("%s\t%s\t\t%d\n",s[i].symbol,s[i].type,s[i].length);

}

$ cc symb.c -o symbout
```

Execution :

1. Navigate to your compiled program directory.
2. Run `./symb.out` in terminal.

ex-3 implementation a symbol table that involves insertion, deletion, search and modify operations using C language

lex.l

```
%{
/* program to recognize a C program */
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 |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
goto {
   printf("\n\t%s is a KEYWORD\n", yytext);
}

"/*" {
   COMMENT = 1;
   printf("\n\t%s is a COMMENT\n", yytext);
}

"*/" {
   COMMENT = 0;
```

```
   printf("\n\t%s is a COMMENT\n", yytext);
}

{identifier}\( {
   if (!COMMENT)
      printf("\n\nFUNCTION\n\t%s", yytext);
}

\{ {
   if (!COMMENT)
      printf("\n BLOCK BEGINS");
}

\} {
   if (!COMMENT)
      printf("\n BLOCK ENDS");
}

{identifier}(\[[0-9]*\])? {
   if (!COMMENT)
      printf("\n\t%s IDENTIFIER", yytext);
}

\".*\" {
   if (!COMMENT)
      printf("\n\t%s is a STRING", yytext);
}

[0-9]+ {
   if (!COMMENT)
      printf("\n\t%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) {
    if (argc > 1) {
        FILE *file;
        file = fopen(argv[1], "r");
        if (!file) {
            printf("could not open %s \n", argv[1]);
            exit(0);
        }
        yyin = file;
    }
    yylex();
    printf("\n\n");
    return 0;
}

int yywrap() {
    return 1;
}
```

test.c

```
#include<stdio.h>
main()
{
int a,b;
}
```

```
$ flex lex1.l
$ cc lex.yy.c -o lex.out
```

Execution :

1. Navigate to your compiled program directory.
2. Run `./lex.out test.c` in term

Ex.No.4 - Use YACC tool to recognize a valid arithmetic expression that uses basic arithmetic operators[+, -, *, /].

exp.y

```
%{
/* validate simple arithmetic expression */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
```

```
%}

%token num let
%left '+' '-'
%left '*' '/'

%%

stmt: stmt '\n' {
    printf("\n Valid \n");
    exit(0);
}
| expr
|
| error '\n' {
    printf("\n Invalid \n");
    exit(0);
}
;

expr: num
| let
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '(' expr ')'
;

%%

int main() {
    printf(" Enter an expression to validate: ");
    yyparse();
}

int yylex() {
    int ch;
    while ((ch = getchar()) == ' ');
    if (isdigit(ch))
        return num;
    if (isalpha(ch))
        return let;
    return ch;
}

void yyerror(char *s) {
    printf("%s", s);
}

$ bison exp.y
$ cc exp.tab.c -o exp.out
```

```

Execution :

1. Navigate to your compiled program directory.
2. Run `./exp.out` in terminal.

# Ex.No.5 - Design a program to recognize a valid variable which starts with an alphabet followed by any number of digits or alphabets using YACC tool

varaibleacc.y

```
%{
/* YACC program to recognize valid variable, which starts with a letter,
   followed by any number of letters or digits. */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
%}

%token let dig

%%

TERM: XTERM '\n' {
    printf("\nAccepted\n");
    exit(0);
}
| error {
    yyerror("\nRejected");
    exit(0);
}
;

XTERM: XTERM let
| XTERM dig
| let
;

%%

int main() {
    printf("Enter a variable: ");
    yyparse();
}

int yylex() {
    char ch;
    while ((ch = getchar()) == ' ');
    if (isalpha(ch))
        return let;
    if (isdigit(ch))
```
```

```
        return dig;
    return ch;
}

void yyerror(char *s) {
    printf("%s", s);
}


$ bison variableyacc.y
$ cc variableyacc.tab.c -o variableyacc.out
```

Execution :

1. Navigate to your compiled program directory.
2. Run `./variableyacc.out` in terminal.

Ex.No.6 - Use LEX and YACC tools to implement a native calculator

calculator.l

```
%{
#include "y.tab.h"
%}

%%

[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[-+*/()]    { return *yytext; }
\n          { return EOL; }
[ \t]       ; /* Ignore whitespace */

.           { fprintf(stderr, "Error: Invalid character '%s'\n", yytext); }

%%

int yywrap() {
    return 1;
}
```

calculator.y

```
%{
#include <stdio.h>
int yylex();
void yyerror(const char *s);

%}

%token NUMBER
```

```
%token EOL

%%

program: /* empty */
      | program expression EOL { printf("Result: %d\n", $2); }
      ;

expression: NUMBER
        | expression '+' expression { $$ = $1 + $3; }
        | expression '-' expression { $$ = $1 - $3; }
        | expression '*' expression { $$ = $1 * $3; }
        | expression '/' expression {
                                    if ($3 == 0) {
                                       yyerror("Error: Division by zero\n");
                                       $$ = 0; // Returning a default value
                                    } else {
                                       $$ = $1 / $3;
                                    }
                                   }
        | '(' expression ')' { $$ = $2; }
        ;

%%

void yyerror(const char *s) {
   fprintf(stderr, "%s", s);
}

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

```
$ lex calculator.l
$ yacc -d calculator.y
$ gcc lex.yy.c y.tab.c -o calculator.out -ll
```

Ex.No.7 - Design a program to generate a three-address code from a given arithmetic expression

gen.c

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

void pm();
void plus();
void div_op();
```

```
void strrev(char *str);

int i, ch, j, l;
char ex[10], ex1[10], exp1[10], ex2[10];

int main() {
   while (1) {
      printf("\n 1. Assignment\n 2. Arithmetic\n 3. Exit\n ENTER THE CHOICE:");
      scanf("%d", &ch);

      switch (ch) {
         case 1:
            printf("\n Enter the e#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void pm();
void plus();
void div_op();
void strrev(char *str);

int i, ch, j, l;
char ex[10], ex1[10], exp1[10], ex2[10];

int main() {
   while (1) {
      printf("\n 1. Assignment\n 2. Arithmetic\n 3. Exit\n ENTER THE CHOICE:");
      scanf("%d", &ch);

      switch (ch) {
         case 1:
            printf("\n Enter the expression with assignment operator:");
            scanf("%s", ex1);
            l = strlen(ex1);
            ex2[0] = '\0';
            i = 0;

            while (ex1[i] != '=') {
               i++;
            }

            strncat(ex2, ex1, i);
            strrev(ex1);
            exp1[0] = '\0';
            strncat(exp1, ex1, l - (i + 1));
            strrev(exp1);
            printf("3 address code:\n temp=%s \n %s=temp\n", exp1, ex2);
            break;

         case 2:
            printf("\n Enter the expression with arithmetic operator:");
            scanf("%s", ex);
```

```
            strcpy(ex1, ex);
            l = strlen(ex1);
            exp1[0] = '\0';

            for (i = 0; i < l; i++) {
               if (ex1[i] == '+' || ex1[i] == '-') {
                  if (ex1[i + 2] == '/' || ex1[i + 2] == '*') {
                     pm();
                     break;
                  } else {
                     plus();
                     break;
                  }
               } else if (ex1[i] == '/' || ex1[i] == '*') {
                  div_op();
                  break;
               }
            }
            break;

         case 3:
            exit(0);
            break;
      }
   }
   return 0;
}

void pm() {
   strrev(exp1);
   j = l - i - 1;
   strncat(exp1, ex1, j);
   strrev(exp1);
   printf("3 address code:\n temp=%s\n temp1=%c%c temp\n", exp1, ex1[j + 2], ex1[j]);
}

void div_op() {
   strncat(exp1, ex1, i + 2);
   printf("3 address code:\n temp=%s\n temp1=temp%c%c\n", exp1, ex1[l + 2], ex1[i + 3]);
}

void plus() {
   strncat(exp1, ex1, i + 2);
   printf("3 address code:\n temp=%s\n temp1=temp%c%c\n", exp1, ex1[l + 2], ex1[i + 3]);
}

// Tool

void strrev(char *str) {
   int start = 0;
   int end = strlen(str) - 1;
```

```
    while (start < end) {
        char temp = str[start];
        str[start] = str[end];
        str[end] = temp;

        start++;
        end--;
    }
}xpression with assignment operator:");
                scanf("%s", ex1);
                l = strlen(ex1);
                ex2[0] = '\0';
                i = 0;

                while (ex1[i] != '=') {
                    i++;
                }

                strncat(ex2, ex1, i);
                strrev(ex1);
                exp1[0] = '\0';
                strncat(exp1, ex1, l - (i + 1));
                strrev(exp1);
                printf("3 address code:\n temp=%s \n %s=temp\n", exp1, ex2);
                break;

            case 2:
                printf("\n Enter the expression with arithmetic operator:");
                scanf("%s", ex);
                strcpy(ex1, ex);
                l = strlen(ex1);
                exp1[0] = '\0';

                for (i = 0; i < l; i++) {
                    if (ex1[i] == '+' || ex1[i] == '-') {
                        if (ex1[i + 2] == '/' || ex1[i + 2] == '*') {
                            pm();
                            break;
                        } else {
                            plus();
                            break;
                        }
                    } else if (ex1[i] == '/' || ex1[i] == '*') {
                        div_op();
                        break;
                    }
                }
                break;

            case 3:
                exit(0);
                break;
```

```
        }
    }
    return 0;
}

void pm() {
    strrev(exp1);
    j = l - i - 1;
    strncat(exp1, ex1, j);
    strrev(exp1);
    printf("3 address code:\n temp=%s\n temp1=%c%c temp\n", exp1, ex1[j + 2], ex1[j]);
}

void div_op() {
    strncat(exp1, ex1, i + 2);
    printf("3 address code:\n temp=%s\n temp1=temp%c%c\n", exp1, ex1[l + 2], ex1[i + 3]);
}

void plus() {
    strncat(exp1, ex1, i + 2);
    printf("3 address code:\n temp=%s\n temp1=temp%c%c\n", exp1, ex1[l + 2], ex1[i + 3]);
}

// Tool

void strrev(char *str) {
    int start = 0;
    int end = strlen(str) - 1;

    while (start < end) {
        char temp = str[start];
        str[start] = str[end];
        str[end] = temp;

        start++;
        end--;
    }
}
```

Ex.No.8 - Implement a simple type checker that checks the scope of the variables and semantic errors sfrom the given statement

typeChecker.c

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


char* type(char[], int);
```

```
void main() {
   char a[10], b[10], mess[20], mess1[20];
   int i, l;
   printf("\n\nint a, b;\n\nint c = a + b\n");
   printf("\n\nEnter a value for a : ");
   scanf("%s", a);
   l = strlen(a);
   printf("\na is : ");
   strcpy(mess, type(a, l));
   printf("%s", mess);
   printf("\n\nEnter a value for b : ");
   scanf("%s", b);
   l = strlen(b);
   printf("\nb is : ");
   strcpy(mess1, type(b, l));
   printf("%s", mess1);

   if( strcmp(mess, "int") == 0 && strcmp(mess1, "int") == 0) {
      printf("\n\nNo Type Error");
   }
   else{
      printf("\n\nType Error");
   }

}


char* type(char x[], int m) {
   int i;
   static char mes[20];

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

      if (isalpha(x[i])) {
         strcpy(mes, "AplhaNumeric");
         goto x;
      }

      else if ( x[i] == '.' ) {
         strcpy(mes, "float");
         goto x;
      }

      strcpy(mes, "int");
      x:;

   }

   return mes;
}
```

Ex.No.9 - Develop a program that optimizes the given input block using code optimization techniques

simpleCode.c

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

struct op {
    char l;
    char r[20];
} op[10], pr[10];

int main() {
    int a, i, k, j, n, z = 0, m, q;
    char *p, *l;
    char temp, t;
    char *tem;

    printf("Enter number of values: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("Left: ");
        op[i].l = getchar();
        op[i].l = getchar();
        printf("\tright: ");
        scanf("%s", op[i].r);
    }


    printf("\nIntermediate Code:\n");

    for (i = 0; i < n; i++) {
        printf("%c=", op[i].l);
        printf("%s\n", op[i].r);
    }

    for (i = 0; i < n - 1; i++) {
        temp = op[i].l;

        for (j = 0; j < n; j++) {
            p = strchr(op[j].r, temp);

            if (p) {
                pr[z].l = op[i].l;
                strcpy(pr[z].r, op[i].r);
                z++;
            }
        }
    }
```

```
pr[z].l = op[n - 1].l;
strcpy(pr[z].r, op[n - 1].r);
z++;

printf("\nAfter Dead code Elimination:\n");

for (k = 0; k < z; k++) {
   printf("%c\t=", pr[k].l);
   printf("%s\n", pr[k].r);
}

for (m = 0; m < z; m++) {
   tem = pr[m].r;

   for (j = m + 1; j < z; j++) {
      p = strstr(tem, pr[j].r);

      if (p) {
         t = pr[j].l;
         pr[j].l = pr[m].l;

         for (i = 0; i < z; i++) {
            l = strchr(pr[i].r, t);

            if (l) {
               a = l - pr[i].r;
               pr[i].r[a] = pr[m].l;
            }
         }
      }
   }
}

printf("\nEliminate Common Expression:\n");

for (i = 0; i < z; i++) {
   printf("%c\t=", pr[i].l);
   printf("%s\n", pr[i].r);
}

for (i = 0; i < z; i++) {
   for (j = i + 1; j < z; j++) {
      q = strcmp(pr[i].r, pr[j].r);

      if ((pr[i].l == pr[j].l) && !q) {
         pr[i].l = '\0';
         strcpy(pr[i].r, "");
      }
   }
}

printf("\nOptimized code: \n");
```

```
    for (i = 0; i < z; i++) {
        if (pr[i].l != '\0') {
            printf("%c=", pr[i].l);
            printf("%s\n", pr[i].r);
        }
    }

    return 0;
}
```

Ex.No.10 - Given an intermediate code as an input. Develop a program that generates the machine code from the given input

code.c

```
#include <stdio.h>

int main() {
    int n, i, j;
    char a[50][50];

    printf("\nEnter the number of intermediate codes: ");
    scanf("%d", &n);

    getchar();

    for (i = 0; i < n; i++) {
        printf("Enter the three address code %d: ", i + 1);

        for (j = 0; j < 6; j++) {
            a[i][j] = getchar();
        }
        getchar();
    }

    printf("\nThe Generated code:\n");

    for (i = 0; i < n; i++) {
        printf("\nMOV %c,R%d", a[i][3], i);

        if (a[i][4] == '-') {
            printf("\nSUB %c,R%d", a[i][5], i);
        } else if (a[i][4] == '+') {
            printf("\nADD %c,R%d", a[i][5], i);
        } else if (a[i][4] == '*') {
            printf("\nMUL %c,R%d", a[i][5], i);
        } else if (a[i][4] == '/') {
            printf("\nDIV %c,R%d", a[i][5], i);
        }

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

```
        printf("\n");
    }

    return 0;
}
```

Ex.No.11 - Generate a valid pattern that recognizes all statements that begins with an Upper-Case Letter followed by five digits or alphabets. Use a YACC tool to do the same.

Uppercase.y

```
%{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
%}

%token UPPER DIGIT

%%

statement: UPPER sequence {
    printf("Accepted\n");
    exit(0);
}
| error {
    yyerror("Rejected\n");
    exit(0);
}
;

sequence: sequence UPPER
| sequence DIGIT
| UPPER
| DIGIT
;

%%

int main() {
    printf("Enter a statement: ");
    yyparse();
}

int yylex() {
    char ch;
    while ((ch = getchar()) == ' ');

    if (isupper(ch))
        return UPPER;

    if (isdigit(ch))
```

```
        return DIGIT;

    return ch;
}

void yyerror(char *s) {
    printf("");
}
```

```
$ bison uppercase_pattern.y
$ cc uppercase_pattern.tab.c -o uppercase_pattern.o
```

Ex.No.12 - Design a lexical analyzer that identifies comments, operators and keywords from a given expression

lex1.l

```
%{
/* program to recognize a C program */
int COMMENT = 0;
%}

identifier [a-zA-Z][a-zA-Z0-9]*

%%

#.* { printf(""); }


int |
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
goto {
    printf("\n\t%s is a KEYWORD\n", yytext);
}
```

```
"/*" {
   COMMENT = 1;
   printf("\n\t%s is a COMMENT\n", yytext);
}

"*/" {
   COMMENT = 0;
   printf("\n\t%s is a COMMENT\n", yytext);
}


{identifier}(\[[0-9]*\])? {
   if (!COMMENT)
      printf("\n\t%s IDENTIFIER", 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) {
   if (argc > 1) {
      FILE *file;
      file = fopen(argv[1], "r");
      if (!file) {
         printf("could not open %s \n", argv[1]);
         exit(0);
      }
      yyin = file;
   }
   yylex();
```

```
   printf("\n\n");
   return 0;
}

int yywrap() {
   return 1;
}
```

test.c

```
#include<stdio.h>
main()
{
int a,b;
a = b;
}
```

execution -

```
$ flex lex1.l
$ cc lex.yy.c -o lex.out
```

Ex.No.13 - Develop a program to recognize a valid control structues syntax of C language (For loop, while loop, if else, if-else-if, switch-case, etc.).

lex1.l

```
%{
/* program to recognize a C program */
int COMMENT = 0;
%}

identifier [a-zA-Z][a-zA-Z0-9]*

%%

#.* { printf(""); }


while |
for |
do |
if |
break |
continue |
switch |
case |
else |
goto {
   printf("\n\t%s is a CONTROL STATEMENT\n", yytext);
}

"/*" {
```

```
   COMMENT = 1;
   printf("\n\t%s is a COMMENT\n", yytext);
}

"*/" {
   COMMENT = 0;
   printf("\n\t%s is a COMMENT\n", yytext);
}



\)(\;)? {
   if (!COMMENT)
      printf("\n\t");
   ECHO;
   printf("\n");
}

\( ECHO;

%%

int main(int argc, char **argv) {
   if (argc > 1) {
      FILE *file;
      file = fopen(argv[1], "r");
      if (!file) {
         printf("could not open %s \n", argv[1]);
         exit(0);
      }
      yyin = file;
   }
   yylex();
   printf("\n\n");
   return 0;
}

int yywrap() {
   return 1;
}
```

test.c

```
#include<stdio.h>
int main() {
          int i;
          for(i=0; i<10; i++){
                    printf("%d", i);
          }
}
```

exec-

```
$ flex lex1.l
$ cc lex.yy.c -o lex.out
```

Ex.No.14 - Develop a Lex Program to find out the total number of vowels and consonants from the given input string.

Lex1.l

```
%{
int vowelCount = 0;
int consonantCount = 0;
%}

%%
[aeiouAEIOU]    { vowelCount++; }
[a-zA-Z]        { consonantCount++; }
.               ; // Ignore other characters

%%

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Usage: %s <input_string>\n", argv[0]);
        return 1;
    }

    yy_scan_string(argv[1]);
    yylex();

    printf("Total number of vowels: %d\n", vowelCount);
    printf("Total number of consonants: %d\n", consonantCount);

    return 0;
}
```

```
$ flex lex1.l
$ cc lex.yy.c -o lex.out -ll
```

OUTPUT:

```
`./lex.out Hello_World`
```
```
Total number of vowels: 3
Total number of consonants: 7
```

Ex.No.15 - Develop a program to generate machine code from a given postfix notation,

gen.c

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

```
#include <stdlib.h>
#include <ctype.h>

// Stack implementation for operands
#define MAX_STACK_SIZE 50

typedef struct {
    int top;
    char items[MAX_STACK_SIZE];
} Stack;

void push(Stack *stack, char item) {
    if (stack->top == MAX_STACK_SIZE - 1) {
        printf("Stack Overflow\n");
        exit(EXIT_FAILURE);
    }
    stack->items[++(stack->top)] = item;
}

char pop(Stack *stack) {
    if (stack->top == -1) {
        printf("Stack Underflow\n");
        exit(EXIT_FAILURE);
    }
    return stack->items[(stack->top)--];
}

// Function to generate machine code from postfix notation
void generateMachineCode(char postfix[]) {
    Stack stack = { .top = -1 };
    int i = 0;
    int regCount = 0;

    while (postfix[i] != '\0') {
        char symbol = postfix[i];

        if (isalnum(symbol)) {
            push(&stack, symbol);
        } else {
            char operand2 = pop(&stack);
            char operand1 = pop(&stack);

            printf("MOV %c,R%d\n", operand1, regCount);
            regCount++;

            switch (symbol) {
                case '+':
                    printf("ADD %c,R%d\n", operand2, regCount);
                    break;
                case '-':
                    printf("SUB %c,R%d\n", operand2, regCount);
                    break;
```

```
            case '*':
                printf("MUL %c,R%d\n", operand2, regCount);
                break;
            case '/':
                printf("DIV %c,R%d\n", operand2, regCount);
                break;
            default:
                printf("Invalid operator\n");
                exit(EXIT_FAILURE);
        }

        push(&stack, 'R' + regCount - 1);
    }

    i++;
  }

  printf("Machine code generation completed.\n");
}

int main() {
  char postfix[50];

  printf("Enter the postfix expression: ");
  scanf("%s", postfix);

  generateMachineCode(postfix);

  return 0;
}
```

$ cc gen.c -o gen.out

OUTPUT:

`./gen.out`
```
Enter the postfix expression: ABC*+
MOV B,R0
MUL C,R1
MOV A,R1
ADD R,R2
Machine code generation completed.
```

Ex.No.16 - Write a LEX program to scan reserved words, variables and operators of C languag

lex.l

```
%{
/* program to recognize a C program */
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 |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |
goto {
    printf("\n\t%s is a RESERVED WORD\n", yytext);
}

"/*" {
    COMMENT = 1;
    printf("\n\t%s is a COMMENT\n", yytext);
}

"*/" {
    COMMENT = 0;
    printf("\n\t%s is a COMMENT\n", yytext);
}

[_a-zA-Z][_a-zA-Z0-9]* {
    printf("Variable: %s\n", yytext);
}

\( 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) {
   if (argc > 1) {
      FILE *file;
      file = fopen(argv[1], "r");
      if (!file) {
         printf("could not open %s \n", argv[1]);
         exit(0);
      }
      yyin = file;
   }
   yylex();
   printf("\n\n");
   return 0;
}

int yywrap() {
   return 1;
}
```

test.c

```
#include<stdio.h>

int main() {
          int a, b;
          a = 10;
          b = 12;
          a = b+c;
}
```

```
$ flex lex1.l
$ cc lex.yy.c -o lex.out
```

output :

```
#include<stdio.h> is a PREPROCESSOR DIRECTIVE


       int is a RESERVED WORD
 Variable: main
() {
```

```
        int is a RESERVED WORD
 Variable: a
, Variable: b
;
        Variable: a

        = is an ASSIGNMENT OPERATOR 10;
        Variable: b

        = is an ASSIGNMENT OPERATOR 12;
        Variable: a

        = is an ASSIGNMENT OPERATOR Variable: b
+Variable: c
;
}
```

Ex.No.17 - Develop a program in C that converts the given three address code into assembly language statements

gen.c

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

typedef struct {
    char result;
    char operand1;
    char operand2;
} ThreeAddressCode;

void convertToAssembly(ThreeAddressCode *code, int numInstructions) {
    for (int i = 0; i < numInstructions; i++) {
        printf("; Assembly code for three-address code %d\n", i + 1);

        char result = code[i].result;
        char operand1 = code[i].operand1;
        char operand2 = code[i].operand2;

        printf("MOV %c, %c\n", operand1, result);
        printf("ADD %c, %c, %c\n", operand2, result, result);

        printf("\n");
    }
}

int main() {
    int numInstructions;

    printf("Enter the number of three-address code instructions: ");
    scanf("%d", &numInstructions);
```

```
    ThreeAddressCode *code = (ThreeAddressCode *)malloc(numInstructions *
sizeof(ThreeAddressCode));

    printf("Enter three-address code:\n");
    for (int i = 0; i < numInstructions; i++) {
        printf("Instruction %d: ", i + 1);
        scanf(" t%c = t%c + t%c", &code[i].result, &code[i].operand1, &code[i].operand2);

        // Clear the input buffer
        while (getchar() != '\n');
    }

    printf("\nAssembly code:\n");
    convertToAssembly(code, numInstructions);

    free(code);

    return 0;
}
```

OUTPUT:

```
`./gen.out`
```
```
Enter the number of three-address code instructions: 3
Enter three-address code:
Instruction 1: t1 = a + b
Instruction 2: t2 = c + d
Instruction 3: t3 = t1 + t2

Assembly code:
; Assembly code for three-address code 1
MOV , 1
ADD , 1, 1

; Assembly code for three-address code 2
MOV , 2
ADD , 2, 2

; Assembly code for three-address code 3
MOV 1, 3
ADD 2, 3, 3
```

Ex.No.18 - Develop a C program to eliminate left recursion from a grammar

eli.c

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

```
#define MAX_RULES 10
#define MAX_SYMBOLS 10

char nonTerminals[MAX_RULES];
char productions[MAX_RULES][MAX_SYMBOLS][MAX_SYMBOLS];

void eliminateLeftRecursion(char nonTerminal, int ruleCount) {
    char newNonTerminal = nonTerminal + 1;

    // Create new productions without left recursion
    for (int i = 0; i < ruleCount; i++) {
        if (productions[i][0][0] == nonTerminal) {
            // A -> Aa | b becomes A -> bA'
            printf("%c -> ", nonTerminal);
            for (int j = 1; productions[i][j][0] != '\0'; j++) {
                printf("%c", productions[i][j][0]);
            }
            printf("%c'\n", newNonTerminal);

            // A' -> aA' | ε
            printf("%c' -> ", newNonTerminal);
            for (int j = 1; productions[i][j][0] != '\0'; j++) {
                printf("%c", productions[i][j][0]);
            }
            printf("%c' | ε\n", newNonTerminal);
        } else {
            // Productions without left recursion remain unchanged
            printf("%c -> ", nonTerminal);
            for (int j = 0; productions[i][j][0] != '\0'; j++) {
                printf("%c", productions[i][j][0]);
            }
            printf("\n");
        }
    }
}

int main() {
    int ruleCount;

    printf("Enter the number of rules: ");
    scanf("%d", &ruleCount);

    printf("Enter the non-terminals:\n");
    for (int i = 0; i < ruleCount; i++) {
        scanf(" %c", &nonTerminals[i]);
    }

    printf("Enter the productions:\n");
    for (int i = 0; i < ruleCount; i++) {
        printf("%c -> ", nonTerminals[i]);

        int symbolCount = 0;
```

```
        while (1) {
            scanf(" %s", productions[i][symbolCount]);
            if (productions[i][symbolCount][strlen(productions[i][symbolCount]) - 1] == ';') {
                productions[i][symbolCount][strlen(productions[i][symbolCount]) - 1] = '\0';
                break;
            }
            symbolCount++;
        }
    }

    printf("\nGrammar after eliminating left recursion:\n");

    for (int i = 0; i < ruleCount; i++) {
        eliminateLeftRecursion(nonTerminals[i], ruleCount);
    }

    return 0;
}
```

OUTPUT:

```
`./el.out`
```
```
Enter the number of rules: 1
Enter the non-terminals:
S
Enter the productions:
S -> S a | S b | c ;

Grammar after eliminating left recursion:
S -> a|Sb|cT'
T' -> a|Sb|cT' | ε
```

Ex.No.19 - Develop a program in C that generates an abstract syntax tree from a given arithmetic expression

gen.c

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

// Node structure for the abstract syntax tree
typedef struct TreeNode {
    char data;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;

// Function to create a new node
TreeNode* createNode(char data) {
```

```
    TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to build the abstract syntax tree
TreeNode* buildAST(char *expression, int *index) {
    TreeNode *root = NULL;
    TreeNode *currentNode = NULL;

    while (expression[*index] != '\0') {
        if (expression[*index] >= '0' && expression[*index] <= '9') {
            // Operand
            currentNode = createNode(expression[*index]);
        } else {
            // Operator
            TreeNode *leftOperand = currentNode;
            currentNode = createNode(expression[*index]);

            (*index)++; // Move to the next character

            // Recursively build the right operand
            currentNode->right = buildAST(expression, index);
            currentNode->left = leftOperand;
        }

        (*index)++; // Move to the next character
    }

    return currentNode;
}

// Function to print the abstract syntax tree (in-order traversal)
void printAST(TreeNode *root) {
    if (root != NULL) {
        printAST(root->left);
        printf("%c ", root->data);
        printAST(root->right);
    }
}

int main() {
    char expression[50];

    printf("Enter an arithmetic expression: ");
    scanf("%s", expression);

    int index = 0;
    TreeNode *root = buildAST(expression, &index);
```

```
    printf("Abstract Syntax Tree (in-order traversal):\n");
    printAST(root);

    return 0;
}
```

OUTPUT:

`./gen.out`

```
Enter an arithmetic expression: 3 + 4 * (5 - 2)
Abstract Syntax Tree (in-order traversal):
3
```

Ex.No.20 - Develop a top-down parser which generates a parsing table with no backtracking

gen.c

```
#include <stdio.h>
#include <ctype.h>

// Global variables
char input[50];
int currentTokenIndex = 0;
int errorFlag = 0;

// Function to match a terminal symbol
void match(char expectedToken) {
    if (input[currentTokenIndex] == expectedToken) {
        currentTokenIndex++;
    } else {
        printf("Error: Expected '%c', found '%c'\n", expectedToken, input[currentTokenIndex]);
        errorFlag = 1;
    }
}

// Function for non-terminal E
void E();

// Function for non-terminal T
void T();

// Function for non-terminal F
void F();

int main() {
    printf("Enter an arithmetic expression: ");
    scanf("%s", input);

    // Reset global variables
```

```
    currentTokenIndex = 0;
    errorFlag = 0;

    // Start parsing
    E();

    // Check for successful parsing
    if (!errorFlag && input[currentTokenIndex] == '\0') {
        printf("Parsing Successful: Valid Arithmetic Expression\n");
    } else {
        printf("Parsing Failed: Invalid Arithmetic Expression\n");
    }

    return 0;
}

// Function for non-terminal E
void E() {
    T();
    while (input[currentTokenIndex] == '+' || input[currentTokenIndex] == '-') {
        char op = input[currentTokenIndex];
        match(op);
        T();
    }
}

// Function for non-terminal T
void T() {
    F();
    while (input[currentTokenIndex] == '*' || input[currentTokenIndex] == '/') {
        char op = input[currentTokenIndex];
        match(op);
        F();
    }
}

// Function for non-terminal F
void F() {
    if (isdigit(input[currentTokenIndex])) {
        match(input[currentTokenIndex]);
    } else if (input[currentTokenIndex] == '(') {
        match('(');
        E();
        match(')');
    } else {
        printf("Error: Unexpected token '%c'\n", input[currentTokenIndex]);
        errorFlag = 1;
    }
}
```

OUTPUT:

`./gen.out`
```
Enter an arithmetic expression: 3 + 4 * (5 - 2)
Parsing Successful: Valid Arithmetic Expression
```

not for educational purpose :)