Compiler Design Laboratory

Experiment-1

Write a C program to identify different types of Tokens in a given Program.

Source Code:

```
#include<stdio.h>
#include<string.h>
#include"stringvalidator.c"

int main(){
    char input[100],token[1][20];
    int i=0,j=0;
    printf("Enter any expression : ");
    scanf("%[^\n]s",input);
    printf("Tokens breakdown : \n");
    while(input[i]!='\0'){
        if(isalnum(input[i])){
            while(isalnum(input[i])){
                token[0][j++]=input[i++];
                token[0][j]='\0';
            }
        }
        else if(inChar(input[i])){
            while(inChar(input[i])){
                token[0][j++]=input[i++];
                token[0][j]='\0';
            }
        }
        else if(isspace(input[i]))
            i++;
        if(j>0){
            if(in(token[0],"keywords"))
                printf("Keyword : %s\n",token[0]);<br>
            else if(in(token[0],"operators"))
                printf("Operator : %s\n",token[0]);
            else if(in(token[0],"specialSymbols"))
                printf("Special Symbol : %s\n",token[0]);
            else
                printf("Identifier/Constant : %s\n",token[0]);
            j=0;
        }
    }
    return 0;
}
```

Contents of stringvalidator.c:

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

char keywords[][32]=
{"auto","break","case","char","const","continue","default","do",
```

```
"double","else","enum","extern","float","for","goto","if",

"int","long","register","return","short","signed","sizeof","static",

"struct","switch","typedef","union","unsigned","void","volatile","while"};
char operators[][38]={".","->","++","--","+","-","!","~","*","&",
                      "/","%","<<",">>","<","<=",">",">=","==","!=",
                      "^","|","&&","||","?",":","=","+=","-=","*=",
                      "/=","%=","&=","^=","|=","<<=",">>=",","};
char specialSymbols[][8]={"[","]","{","}","(",")",";","#"};
char symbols[]=".,->+=~!#%^&*(){}[]|/;:?<>";

int inChar(char tk){
    int i;
    for(i=0;i<strlen(symbols);i++)
        if(tk==symbols[i])
            return 1;
    return 0;
}

int in(char * token,char * bin){
    int i;
    if(strcmp(bin,"keywords")==0){
        for(i=0;i<sizeof(keywords)/sizeof(keywords[0]);i++)
            if(strcmp(token,keywords[i])==0)
                return 1;
        return 0;
    }
    else if(strcmp(bin,"operators")==0){
        for(i=0;i<sizeof(operators)/sizeof(operators[0]);i++)
            if(strcmp(token,operators[i])==0)
                return 1;
        return 0;
    }
    else if(strcmp(bin,"specialSymbols")==0){
        for(i=0;i<sizeof(specialSymbols)/sizeof(specialSymbols[0]);i++)
            if(strcmp(token,specialSymbols[i])==0)
                return 1;
        return 0;
    }
    return 0;
}
```

Output:

```
Enter any expression : int a=3 ; a++ ; a=b-c++ ;
Tokens breakdown :
Keyword : int
Identifier/Constant : a
Operator : =
Identifier/Constant : 3
Special Symbol : ;
Identifier/Constant : a
Operator : ++
Special Symbol : ;
Identifier/Constant : a
Operator : =
Identifier/Constant : b
Operator : -
Identifier/Constant : c
```

```
Operator : ++
Special Symbol : ;
```

Experiment -2

Write a Lex Program to implement a Lexical Analyzer using Lex tool

Source Code:

```
%{
    #include<stdio.h>
    int Upper=0;
    int Lower=0;
    int Number=0;
    int i,op=0;
%}
UPPER [A-Z]
LOWER [a-z]
ADD +
SUBTRACT -
MULTIPLY "*"
DIVIDE "/"
EXPONENT "**"
NUMBER [0-9]+
%%

{UPPER} Upper++;
{LOWER} Lower++;
{NUMBER} Number++; calc();
{ADD}       op=1;
{SUBTRACT}  op=2;
{MULTIPLY}  op=3;
{DIVIDE}    op=4;
{EXPONENT}  op=5;
\n  {printf("\nThe Answer is : %f\n",a);} return 0;
%%
calc(){
    if(op==0){
        a=atof(yytext);
    }
    else{
        b=atof(yytext);
        switch(op){
            case 1: a=a+b;
                break;
            case 2: a=a-b;
                break;
            case 3: a=a*b;
                break;
            case 4: a=a/b;
                break;
            case 5: for(i=a;b>1;b--) a*=i;
                break;
        }
    op=0;
    }
}
yywrap(){}
main()
```

```
{
    printf("Enter a string : ");
    yyin=stdin;
    yylex();
    printf("\nUppercase : %d, Lowercase : %d, and Digits :
%d",Upper,Lower,Number);
    yylex();
}
```

Output:

```
Enter a string : UcenCSE03

Uppercase : 4, Lowercase : 3, and Digits : 2
```

Experiment - 3

Write a C program to Simulate Lexical Analyzer to validating a given input String.

Source Code:

```
/*

    3. Write a C program to simulate Lexical Analyzer to validating a given input
string.

*/

#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include"stringvalidator.c"

int main(){
    char input[10];
    int id=0,key=0,num=0,valid=0,i=1;
    printf("-------------This program checks if the given string is any of
identifier, keyword, number, operator, or special symbol-------------\n\n");
    printf("Enter a string : ");
    fgets(input,sizeof(input),stdin);
    input[strcspn(input,"\n")]='\0';
    if(input[0]=='_'){
        if(input[i]=='\0'){
            printf("Valid identifier");
            return 0;
        }
        while(input[i]!='\0'){
            if(isdigit(input[i]) || isalpha(input[i]) ||input[i]=='_'){
                id=1;
                valid=1;
            }
            else{
                id=0;
                valid=0;
                break;
            }
            i++;
        }
```

```
    }
    else if(isalpha(input[0])){
        if(in(input,"keywords")){
            key=1;
            valid=1;
        }
        else{
            if(input[i]=='\0'){
                printf("Valid identifier");
                return 0;
            }
            while(input[i]!='\0'){
                if(isdigit(input[i]) || isalpha(input[i]) || input[i]=='_'){
                    id=1;
                    valid=1;
                }
                else{
                    id=0;
                    valid=0;
                    break;
                }
                i++;
            }
        }
    }
    else if(isdigit(input[0])){
        while(input[i]!='\0'){
            if(isdigit(input[i])){
                num=1;
                valid=1;
                break;
            }
            else{
                num=0;
                valid=0;
                break;
            }
            i++;
        }
    }
    else if(input[0]=='"' || input[0]=='\''){
        while(input[i]!='\0'){
            if((input[0]=='"' && input[i]=='"')|| (input[0]=='\''
&&input[i]=='\'')){
                if(input[i+1]=='\0')
                    printf("Valid string");
                else
                    printf("Invalid string");
                return 0;
            }
            i++;
        }
    }
    else if(in(input,"operators")){
        printf("Valid operator");
        return 0;
    }
    else if(in(input,"specialSymbols")){
        printf("Valid special symbol");
        return 0;
    }
```

```
    if(valid){
        if(id)
            printf("Valid identifier");
        else if(key)
            printf("Valid keyword");
        else if(num)
            printf("Valid number");
    }
    else
        printf("Invalid input string");
    return 0;
}
```

Output:

```
-------------This program checks if the given string is any of identifier,
keyword, number, operator, or special symbol-------------

Enter a string : int
Valid keyword
```

Experiment - 4

Write a C program to implement the Brute force technique of Top down Parsing.

Source Code:

```
/*

    Write a C program to implement the Brute force technique of Top down parsing.

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_PROD 10
#define MAX_LEN 50

char productions[MAX_PROD][MAX_LEN];
int num_productions = 0;
char input[MAX_LEN];
int input_pos = 0;

int parse(char non_terminal) {
    int i,j;
    for (i = 0; i < num_productions; i++) {
        if (productions[i][0] == non_terminal) {
            int saved_pos = input_pos;
            for (j = 2; productions[i][j] != '\0'; j++) {
                if (isupper(productions[i][j])) {
                    if (!parse(productions[i][j])) {
                        break;
                    }
                } else if (productions[i][j] == '$') {
                    continue;
                } else {
```

```
                    if (input[input_pos] == productions[i][j]) {
                        input_pos++;
                    } else {
                        break;
                    }
                }
            }
            if (productions[i][j] == '\0') {
                return 1;
            }
            input_pos = saved_pos;
        }
    }
    return 0;
}

int main() {
    int i;
    printf("Enter the number of productions: ");
    scanf("%d", &num_productions);
    getchar(); // Consume newline

    printf("Enter the productions (use $ for epsilon):\n");
    for (i = 0; i < num_productions; i++) {
        fgets(productions[i], MAX_LEN, stdin);
        productions[i][strcspn(productions[i], "\n")] = '\0'; // Remove newline
    }

    printf("Enter the input string: ");
    fgets(input, MAX_LEN, stdin);
    input[strcspn(input, "\n")] = '\0'; // Remove newline

    if (parse(productions[0][0]) && input[input_pos] == '\0') {
        printf("Input string is accepted by the grammar.\n");
    } else {
        printf("Input string is not accepted by the grammar.\n");
    }

    return 0;
}
```

Output:

```
Enter the number of productions: 2
Enter the productions (use $ for epsilon):
S=aSb
S=$
Enter the input string: aabb
Input string is accepted by the grammar.
```

Experiment - 5

Write a C program to implement a Recursive Descent Parser.

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
char input[100];
```

```c
void E();
void EPrime();
void T();
void TPrime();
void F();
int error=0,i=0;
int main() {
    printf("Enter an arithmetic expression : ");
    gets(input);
    input[strcspn(input,"\n")] = '\0';
    E();
    if(strlen(input)==i && error==0) printf("accepted");
    else printf("Rejected");
    return 0;
}

void E(){
    T();
    EPrime();
}

void T(){
    F();
    TPrime();
}

void EPrime(){
    if(input[i]=='+') {
        i++;
        T();
        EPrime();
    }
}

void TPrime(){
    if(input[i]=='*'){
        i++;
        F();
        TPrime();
    }
}

void F(){
    if(input[i]=='('){
        i++;
        E();
        if(input[i]==')') i++;
    }
    else if(isalpha(input[i]) || isalnum(input[i])){
        while(isalpha(input[i]) || isalnum(input[i])) i++;
    }
    else error=1;
}
```

Output:

```
Enter an arithmetic expression : sum+month*interest
accepted
```

Experiment - 6

Write C program to compute the First and Follow Sets for the given Grammar.

Source Code:

```c
#include<stdio.h>
#include<string.h>
#include<ctype.h>
int nop,m=0,p,i=0,j=0;
char prod[10][10],res[10];
void Follow(char c);
void first(char c);
void result(char c);
int main(){
    char c,ch;
    int i,choice;
    printf("enter no.of productions: ");
    scanf("%d",&nop);
    printf("enter production string like E=E+t\n");
    for(i=0;i<nop;i++){
        printf("Enter productions number %d : ",i+1);
        scanf("%s",prod[i]);
    }
    for(i=0;i<nop;i++){
        first(prod[i][0]);
        printf("First(%c)={%s}\n",prod[i][0],res);
        m=0;
    }
    for(i=0;i<nop;i++){
        Follow(prod[i][0]);
        printf("Follow(%c)={%s}\n",prod[i][0],res);
        m=0;
    }
    return 0;
}

void Follow(char c){
    if(prod[0][0]==c){
        result('$');
    }
    for(i=0;i<nop;i++){
        for(j=2;j<strlen(prod[i]);j++){
            if(prod[i][j]==c){
                if(prod[i][j+1]!='\0') first(prod[i][j+1]);
                if(prod[i][j+1]=='\0' && c!=prod[i][0]) Follow(prod[i][0]);
            }
        }
    }
}

void first(char c){
    int k;
    if(!isupper(c)) result(c);
    for(k=0;k<nop;k++){
        if(prod[k][0]==c){
            if(prod[k][2]=='@') Follow(prod[i][0]);
            else if(islower(prod[k][2])) result(prod[k][2]);
            else first(prod[k][2]);
        }
```

```
    }
}

void result(char c){
    int i;
    for(i=0;i<=m;i++){
        if(res[i]==c) return;
    }
    res[m++] = c;
    res[m++] =',';
}
```

Output:

```
enter no.of productions: 3
enter production string like E=E+t
Enter productions number 1 : S=A+B
Enter productions number 2 : A=a
Enter productions number 3 : B=b
First(S)={a,}
First(A)={a,}
First(B)={b,}
Follow(S)={$,}
Follow(A)={+,}
Follow(B)={$,}
```

Experiment - 7

Write a C program for eliminating the left recursion and left factoring of a given grammar

Source Code:

```
#include<stdio.h>
void removeLeftRecursion(char nonTerminal, char alpha[], char beta[]){
    char newNonTerminal = nonTerminal+1;
    printf("New production rules after removing left recursion : \n");
    printf("%c->%s%c\n",nonTerminal,beta,newNonTerminal);
    printf("%c->%s%c|E\n",newNonTerminal,alpha,newNonTerminal);
}

int main(){
    char alpha[20],nonTerminal,beta[20];
    printf("Enter non terminal : ");
    scanf("%c",&nonTerminal);
    printf("Enter left recursive part(alpha) : ");
    scanf("%s",&alpha);
    printf("Enter right recursive part(beta) : ");
    scanf("%s",&beta);
    removeLeftRecursion(nonTerminal,alpha,beta);
}
```

Output:

```
Enter non terminal : A
Enter left recursive part(alpha) : A
Enter right recursive part(beta) : b
New production rules after removing left recursion :
```

```
A->bB
B->AB|E
```

Source Code:

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

int main(){
    char gram[20], part1[20],part2[20],newGram[20],modifiedGram[20];
    int i,j=0,k=0,l=0,pos;
    printf("Enter production : A -> ");
    gets(gram);
    for (i=0;gram[i]!='|';i++,j++)
        part1[j]=gram[i];
    part1[j]='\0';
    i++;

    for (j=0;gram[i]!='\0';i++,j++)
        part2[j]=gram[i];
    part2[j]='\0';

    for (i=0;i<strlen(part1)||i<strlen(part2);i++){
        if(part1[i]==part2[i]){
            modifiedGram[k]=part1[i];
            k++;
            pos=i+1;
        }
    }

    for(i=pos,j=0;part1[i]!='\0';i++,j++)
        newGram[j]=part1[i];
    newGram[j++]='|';
    for(i=pos;part2[i]!='\0';i++,j++)
        newGram[j]=part2[i];
    modifiedGram[k]='X';
    modifiedGram[++k]='\0';
    newGram[j]='\0';
    printf("A -> %s\n",modifiedGram);
    printf("A -> %s",newGram);
    return 0;
}
```

Output:

```
Enter production : A -> aabbc|aabbddc
A -> aabbX
A -> c|ddc
```

Experiment - 8

Write a C program to check the validity of input string using Predictive Parser.

Source Code:

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

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

#define MAX_PROD 10
#define MAX_STACK 100
#define MAX_INPUT 100

char productions[MAX_PROD][20];
int parse_table[26][128];  // Using int for production indices
char stack[MAX_STACK];
int top = -1;
char input[MAX_INPUT];
int input_index = 0;

void push(char c) {
    if (top < MAX_STACK - 1) {
        stack[++top] = c;
    }
}

char pop() {
    if (top >= 0) {
        return stack[top--];
    }
    return '\0';
}

void create_parse_table() {
    int i, j, k;
    char lhs, *rhs, current;

    // Initialize parse table with -1 (error entries)
    for (i = 0; i < 26; i++) {
        for (j = 0; j < 128; j++) {
            parse_table[i][j] = -1;
        }
    }

    // Fill parse table
    for (i = 0; productions[i][0] != '\0'; i++) {
        lhs = productions[i][0];
        rhs = &productions[i][2];

        if (*rhs == '$') { // Epsilon production
            parse_table[lhs - 'A']['$'] = i;
        }
        else {
            char* current = rhs;
            while (*current != '\0') {
                if (islower(*current)) {
                    parse_table[lhs - 'A'][(int)*current] = i;
                    break;
                }
                else {
                    // Follow non-terminal chain
                    char nt = *current;
                    int found = 0;
                    for (j = 0; productions[j][0] != '\0'; j++) {
                        if (productions[j][0] == nt) {
                            if (productions[j][2] != '$' &&
```

```
!isupper(productions[j][2])) {
                                parse_table[lhs - 'A'][(int)productions[j][2]] =
i;
                                found = 1;
                                break;
                            }
                            current = &productions[j][2];
                        }
                    }
                    if (found) break;
                }
                current++;
            }
        }
    }
}

void parse() {
    int i, prod_index;
    char stack_top, current_input;
    char stack_str[MAX_STACK];

    push('$');
    push(productions[0][0]);

    printf("\nParsing Steps:\n");
    printf("%-15s %-15s %s\n", "Stack", "Input", "Action");
    printf("-----------------------------------------\n");

    while (top >= 0) {
        // Build stack string
        for (i = top; i >= 0; i--) {
            stack_str[top - i] = stack[i];
        }
        stack_str[top + 1] = '\0';

        printf("%-15s %-15s ", stack_str, input + input_index);

        stack_top = pop();
        current_input = input[input_index];

        if (stack_top == '$' && current_input == '$') {
            printf("Accept!\n");
            return;
        }

        if (stack_top == current_input) {
            printf("Match '%c'\n", current_input);
            input_index++;
        }
        else if (islower(stack_top)) {
            printf("Error: Expected '%c'\n", stack_top);
            return;
        }
        else {
            prod_index = parse_table[stack_top - 'A'][(int)current_input];
            if (prod_index == -1) {
                printf("Error: No production for %c on '%c'\n", stack_top,
current_input);
                return;
            }
```

```
            printf("Apply %s\n", productions[prod_index]);

            char* rhs = &productions[prod_index][2];
            for (i = strlen(rhs)-1; i >= 0; i--) {
                if (rhs[i] != '$') push(rhs[i]);
            }
        }
    }
}

int main() {
    int num_prod, i;

    printf("Enter number of productions: ");
    scanf("%d", &num_prod);
    getchar();

    printf("\nEnter productions in format LHS=RHS (use $ for epsilon):\n");
    for (i = 0; i < num_prod; i++) {
        printf("Production %d: ", i+1);
        fgets(productions[i], 20, stdin);
        productions[i][strcspn(productions[i], "\n")] = '\0';
    }

    create_parse_table();

    printf("\nEnter input string: ");
    scanf("%s", input);
    strcat(input, "$");

    parse();

    return 0;
}
```

Output:

```
Enter number of productions: 3

Enter productions in format LHS=RHS (use $ for epsilon):
Production 1: S=A+B
Production 2: A=a
Production 3: B=b

Enter input string: a+b

Parsing Steps:
Stack           Input           Action
----------------------------------------
S$              a+b$            Apply S=A+B
A+B$            a+b$            Apply A=a
a+B$            a+b$            Match 'a'
+B$             +b$             Match '+'
B$              b$              Apply B=b
b$              b$              Match 'b'
$               $               Accept!
```

Experiment - 9

Write a C program for implementation of LR parsing algorithm to accept a given input string.

Source Code

```c
#include <stdio.h>
#include <string.h>
#define MAX 100
char stack[MAX];
int top = -1;
char input[MAX];
int ip = 0;
// Push to stack
void push(char c) {
 stack[++top] = c;
}

// Pop from stack
void pop() {
 if (top >= 0)
 top--;
}
// Display stack contents
void displayStack() {
 for (int i = 0; i <= top; i++) {
    printf("%c", stack[i]);
 }
}
// Check if the stack contains a reducible pattern
void reduce() {
 while (1) {
 if (top >= 2 && stack[top] == 'E' && stack[top - 1] == '+' && stack[top - 2] ==
'E') {
 // Reduce E + E -> E
     top -= 2;
     stack[top] = 'E';
     printf("\nReduce: E -> E+E");
 }
 else if (top >= 2 && stack[top] == 'E' && stack[top - 1] == '*' && stack[top - 2]
== 'E') {
 // Reduce E * E -> E
     top -= 2;
     stack[top] = 'E';
     printf("\nReduce: E -> E*E");
 }
 else if (top >= 2 && stack[top] == ')' && stack[top - 1] == 'E' && stack[top - 2]
== '(') {
 // Reduce (E) -> E
     top -= 2;
     stack[top] = 'E';
     printf("\nReduce: E -> (E)");
 }
 else if (top >= 0 && stack[top] == 'i') {
     // Reduce id -> E
     stack[top] = 'E';
     printf("\nReduce: E -> i");
 }
 else {
    break; // No more reductions possible
 }
```

```
 }
}
// LR Parsing function
void parse() {
 printf("\nParsing steps:");
 while (input[ip] != '\0') {
     push(input[ip]); // Shift operation
     printf("\nShift: ");
     displayStack();
     reduce(); // Reduce operation if applicable
     ip++;
     }
 // Final reduction check
 reduce();
 // If final stack contains only 'E', the string is accepted
 if (top == 0 && stack[top] == 'E') {
    printf("\nString Accepted.\n");
 } else {
    printf("\nString Rejected.\n");
 }
}
// Main function
int main() {
 printf("Enter input string (e.g., i+i*i): ");
 scanf("%s", input);
 parse();
 return 0;
}
```

Output:

```
Enter input string (e.g., i+i*i): i+i

Parsing steps:
Shift: i
Reduce: E -> i
Shift: E+
Shift: E+i
Reduce: E -> i
Reduce: E -> E+E
String Accepted.
```

Experiment - 10

Write a C program for implementation of a Shift Reduce Parser using Stack Data Structure to accept a given input string of a given grammar.

Source Code:

Refer your record or internet

Output:

no output

Experiment - 11

Simulate the calculator using LEX and YACC tool.

Source Code:

```
%{
    #include<stdio.h>
    int i,op=0;
    int is_int=0;
    float a,b;
%}

ADD "+"
SUBTRACT "-"
MULTIPLY "*"
DIVIDE "/"
EXPONENT "**"
NUMBER [0-9]+|([0-9]*)"."([0-9]+)
EXIT "exit"
%%

{NUMBER}    calc();
{ADD}       op=1;
{SUBTRACT}  op=2;
{MULTIPLY}  op=3;
{DIVIDE}    op=4;
{EXPONENT}  op=5;
{EXIT}  return 0;
\n      {printf("\nThe Answer is : %f\n",a);}


%%

calc(){
    if(op==0){
        a=atof(yytext);
    }
    else{
        b=atof(yytext);
        switch(op){
            case 1: a=a+b;
                break;
            case 2: a=a-b;
                break;
            case 3: a=a*b;
                break;
            case 4: a=a/b;
                break;
            case 5: for(i=a;b>1;b--) a*=i;
                break;
        }
    op=0;
    }
}
yywrap(){return 1;}
main(int argv,char*argc[]){
    printf("Enter the expressions below!\nAvailable operations are
+,-,*,/,**\nEnter \"exit\" to exit the program\n\n");
    yylex();
}
```

Output:

```
Enter the expressions below!
Available operations are +,-,*,/,**
Enter "exit" to exit the program

3+5

The Answer is : 8.000000
3-5

The Answer is : -2.000000
3*5

The Answer is : 15.000000
exit
```

Experiment - 12

Generate YACC specification for a few syntactic categories.

YACC Specifications:

```
%{
#include <stdio.h>
int regs[26];
int base;
%}

%start list
%union {int a;}
%token <a> DIGIT LETTER
%type <a> expr stat number
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS

%%
list: /* empty */
    | list stat '\n'
    | list error '\n' {yyerrok;}
    ;
stat: expr {printf("%d\n",$1);}
    | LETTER '=' expr {regs[$1]=$3;}
    ;
expr: '(' expr ')' {$$=$2;}
    | expr '*' expr {$$=$1*$3;}
    | expr '/' expr {$$=$1/$3;}
    | expr '%' expr {$$=$1%$3;}
    | expr '+' expr {$$=$1+$3;}
    | expr '-' expr {$$=$1-$3;}
    | expr '&' expr {$$=$1&$3;}
    | expr '|' expr {$$=$1|$3;}
    | '-' expr %prec UMINUS {$$=-$2;}
    | LETTER {$$=regs[$1];}
    | number
```

```
    ;
number: DIGIT {$$=$1; base=($1==0)?8:10;}
    | number DIGIT {$$=base * $1 + $2;}
    ;
%%

int main(){
    while(1){
        printf("> ");
        yyparse();
        yyrestart(yyin);
    }
    return 0;
}

void yyerror(const char *s){
    fprintf(stderr,"Error: %s\n",s);
    valid=0;
}
```

Output:

No output

Experiment - 13

Write a C program for generating the three address code of a given expression/statement.

Source Code:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#define MAX 100
typedef struct {
    char op;
    int prec;
} Operator;
int tempVarCount = 1; // Counter for temporary variables
// Function to get precedence of operators
int precedence(char op) {
     switch (op) {
         case '+': case '-': return 1;
         case '*': case '/': return 2;
         default: return 0;
     }
}
// Stack implementation for operators
char opStack[MAX];
int opTop = -1;
void pushOp(char op) {
    opStack[++opTop] = op;
}
char popOp() {
    return opStack[opTop--];
}
char peekOp() {
    return (opTop >= 0) ? opStack[opTop] : '\0';
```

```
}
// Stack for operands
char operandStack[MAX][10];
int operandTop = -1;
void pushOperand(char *operand) {
    strcpy(operandStack[++operandTop], operand);
}
char* popOperand() {
    return operandStack[operandTop--];
}
// Function to generate TAC
void processOperator() {
     char op = popOp();
     char operand2[10], operand1[10], result[10];
     strcpy(operand2, popOperand());
     strcpy(operand1, popOperand());
     sprintf(result, "t%d", tempVarCount++);

     // Print Three Address Code
     printf("%s = %s %c %s\n", result, operand1, op, operand2);

     // Push result back as operand
     pushOperand(result);
}
void generateTAC(char exp[]) {
     int len = strlen(exp);

     for (int i = 0; i < len; i++) {
         if (isalnum(exp[i])) { // If operand
             char operand[2] = {exp[i], '\0'};
             pushOperand(operand);
        }
        else if (strchr("+-*/", exp[i])) { // If operator
             while (opTop >= 0 && precedence(peekOp()) >= precedence(exp[i])) {
                processOperator();
             }
            pushOp(exp[i]);
        }
         else if (exp[i] == '(') {
            pushOp(exp[i]);
         }
         else if (exp[i] == ')') {
             while (opTop >= 0 && peekOp() != '(') {
                processOperator();
             }
            popOp(); // Remove '('
         }
    }

 // Process remaining operators
     while (opTop >= 0) {
        processOperator();
     }
}
int main() {
     char expression[50];
     // Input: Read an arithmetic expression
     printf("Enter an arithmetic expression (e.g., a+b*c): ");
     scanf("%s", expression);
     // Generate TAC
     printf("\nThree Address Code:\n");
```

```
    generateTAC(expression);
    return 0;
}
```

Output:

```
Enter an arithmetic expression (e.g., a+b*c): a+b*c

Three Address Code:
t1 = b * c
t2 = a + t1
```

Experiment - 14

Write a C program for implementation of a Code Generation Algorithm of a given expression/statement.

Source Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
// Stack structure for operands
typedef struct {
    char data[MAX][MAX];
    int top;
} OperandStack;
// Generate assembly-like code from postfix expression
void generateCode(char *postfix) {
    OperandStack opStack;
    opStack.top = -1;
    int reg = 1; // Register counter
    printf("\nGenerated Assembly-like Code:\n");
    for (int i = 0; postfix[i] != '\0'; i++) {
        if (isalnum(postfix[i])) { // Operand
            char operand[2] = {postfix[i], '\0'};
            strcpy(opStack.data[++opStack.top], operand);
        }
    else { // Operator
        char op2[MAX], op1[MAX], result[MAX];
        strcpy(op2, opStack.data[opStack.top--]);
        strcpy(op1, opStack.data[opStack.top--]);
        printf("MOV R%d, %s\n", reg, op1);
        if (postfix[i] == '+')
            printf("ADD R%d, %s\n", reg, op2);
        else if (postfix[i] == '-')
            printf("SUB R%d, %s\n", reg, op2);
        else if (postfix[i] == '*')
            printf("MUL R%d, %s\n", reg, op2);
        else if (postfix[i] == '/')
            printf("DIV R%d, %s\n", reg, op2);
        sprintf(result, "R%d", reg);
        strcpy(opStack.data[++opStack.top], result);
        reg++;
    }
}
```

```
 printf("RESULT STORED IN %s\n", opStack.data[opStack.top]);
}
// Main function
int main() {
 char infix[MAX], postfix[MAX];
 printf("Enter a postfix arithmetic expression : ");
 scanf("%s", postfix);
 //infixToPostfix(infix, postfix);
 printf("Postfix Expression: %s\n", postfix);
 generateCode(postfix);
 return 0;
}
```

Output:

```
Enter a postfix arithmetic expression : ab+cd*/
Postfix Expression: ab+cd*/

Generated Assembly-like Code:
MOV R1, a
ADD R1, b
MOV R2, c
MUL R2, d
MOV R3, R1
DIV R3, R2
RESULT STORED IN R3
```