

1. Write a C program to identify different types of tokens in a given program.

```
#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);
}

```

Output:

```

'int' IS A KEYWORD
'a' IS A VALID IDENTIFIER
'=' IS AN OPERATOR
'b' IS A VALID IDENTIFIER
'+' IS AN OPERATOR
'1c' IS NOT A VALID IDENTIFIER

```

2. Write a Lex Program to implement a Lexical Analyzer using Lex Tool.

```
% {
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",yytext);}
"/ * " { COMMENT = 1;}
" * / " { COMMENT = 0;}
{ 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 %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 0;
}

```

### ODDREVEN.C

```

#include <stdio.h>
int main()
{
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);
    if(number % 2 == 0)
        printf("%d is even.", number);
    else
        printf("%d is odd.", number);
    return 0;
}

```

## Output:

☐ flex p1.l

flex: command not installed. Multiple versions of this command were found in Nix.

Select one to run (or press Ctrl-C to cancel):

Adding flex to replit.nix

success

fle/nix/store/sm1igy7xz1c54l14mzi562x4km55i7vg-flex-2.6.4

x p1.Detected change in environment, reloading shell...

☐ flex p1.l

☐ gcc lex.yy.c

☐ ./a.out oddreven.c

3. Write a C program to implement the Brute Force Technique of Top Down Parsing.

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

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

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

```
char input[100];
```

```
int inde = 0;
```

```
int error = 0;
```

```
void E();
```

```
void T();
```

```
void F();
```

```
void match(char c) {
```

```
    if (input[inde] == c) {
```

```
        inde++;
```

```
    } else {
```

```
        error = 1;
    }
}
```

```
void E() {
    T();
    if (input[inde] == '+') {
        match('+');
        E();
    }
}
```

```
void T() {
    F();
    if (input[inde] == '*') {
        match('*');
        T();
    }
}
```

```
void F() {
    if (input[inde] == '(') {
        match('(');
        E();
        match(')');
    } else if (input[inde] == 'i') {
        match('i');
    } else {
        error = 1;
    }
}
```



```

    }
}

int main() {
    printf("Enter an arithmetic expression: ");
    fgets(input, 100, stdin);

    // Remove newline character
    input[strcspn(input, "\n")] = '\0';

    E();

    if (error == 0 && inde == strlen(input)) {
        printf("Valid expression\n");
    } else {
        printf("Invalid expression\n");
    }

    return 0;
}

```

Output:

Enter an arithmetic expression: (i+i)\*i

Valid expression

4. Write a C program to implement a Recursive Descent Parser.

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

#define SUCCESS 1
#define FAILED 0

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

const char *cursor;
char string[64];

int main()
{
    puts("Enter the string");
    // scanf("%s", string);
    sscanf("i+(i+i)*i", "%s", string);
    cursor = string;
    puts("");
    puts("Input      Action");
    puts("-----");

    if (E() && *cursor == '\0') {
        puts("-----");
        puts("String is successfully parsed");
        return 0;
    } else {
        puts("-----");
```

```
    puts("Error in parsing String");  
    return 1;  
}  
}
```

```
int E()  
{  
    printf("%-16s E -> T E'\n", cursor);  
    if (T()) {  
        if (Edash())  
            return SUCCESS;  
        else  
            return FAILED;  
    } else  
        return FAILED;  
}
```

```
int Edash()  
{  
    if (*cursor == '+') {  
        printf("%-16s E' -> + T E'\n", cursor);  
        cursor++;  
        if (T()) {  
            if (Edash())  
                return SUCCESS;  
            else  
                return FAILED;  
        } else  
            return FAILED;  
    }
```

```

    } else {
        printf("%-16s E' -> $\n", cursor);
        return SUCCESS;
    }
}

```

```

int T()
{
    printf("%-16s T -> F T'\n", cursor);
    if (F()) {
        if (Tdash())
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}

```

```

int Tdash()
{
    if (*cursor == '*') {
        printf("%-16s T' -> * F T'\n", cursor);
        cursor++;
        if (F()) {
            if (Tdash())
                return SUCCESS;
            else
                return FAILED;
        } else

```

```

        return FAILED;
    } else {
        printf("%-16s T' -> $\n", cursor);
        return SUCCESS;
    }
}

```

```

int F()
{
    if (*cursor == '(') {
        printf("%-16s F -> ( E )\n", cursor);
        cursor++;
        if (E()) {
            if (*cursor == ')') {
                cursor++;
                return SUCCESS;
            } else
                return FAILED;
        } else
            return FAILED;
    } else if (*cursor == 'i') {
        cursor++;
        printf("%-16s F -> i\n", cursor);
        return SUCCESS;
    } else
        return FAILED;
}

```

Output:

/tmp/HKzwdTYnft.o

Enter the string

Input	Action
-------	--------

-----

i+(i+i)*i	E -> T E'
-----------	-----------

i+(i+i)*i	T -> F T'
-----------	-----------

+(i+i)*i	F -> i
----------	--------

+(i+i)*i	T' -> \$
----------	----------

+(i+i)*i	E' -> + T E'
----------	--------------

(i+i)*i	T -> F T'
---------	-----------

(i+i)*i	F -> ( E )
---------	------------

i+i)*i	E -> T E'
--------	-----------

i+i)*i	T -> F T'
--------	-----------

+i)*i	F -> i
-------	--------

+i)*i	T' -> \$
-------	----------

+i)*i	E' -> + T E'
-------	--------------

i)*i	T -> F T'
------	-----------

)*i	F -> i
-----	--------

)*i	T' -> \$
-----	----------

)*i	E' -> \$
-----	----------

*i	T' -> * F T'
----	--------------

	F -> i
--	--------

	T' -> \$
--	----------

	E' -> \$
--	----------

-----

String is successfully parsed

5. Write a C program to compute the First and Follow sets for the given grammar.

```
// C program to calculate the First and
```

```
// Follow sets of a given grammar
```

```
#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;
```

```
    count = 8;
```

```
    // The Input grammar
```

```
    strcpy(production[0], "X=TnS");
```

```
    strcpy(production[1], "X=Rm");
```

```
    strcpy(production[2], "T=q");
```

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

```
    strcpy(production[4], "S=p");
```

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

```
    strcpy(production[6], "R=om");
```

```
    strcpy(production[7], "R=ST");
```

```
    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++;
}
}
}

```

### Output:

First(X) = { q, n, o, p, #, }

First(T) = { q, #, }

First(S) = { p, #, }

First(R) = { o, p, q, #, }

-----

Follow(X) = { \$, }

Follow(T) = { n, m, }

Follow(S) = { \$, q, m, }

Follow(R) = { m, }



6. Write a C program for eliminating Left Recursion and Left Factoring of a given grammar.

Program for Left Recursion:

```
#include<stdio.h>
#include<string.h>
#define SIZE 10
int main ()
{
    char non_terminal;
    char beta,alpha;
    int num;
    char production[10][SIZE];
    int index=3; /* starting of the string following "->" */
    printf("Enter Number of Production : ");
    scanf("%d",&num);
    printf("Enter the grammar as E->E-A :\n");
    for(int i=0;i<num;i++)
    {
        scanf("%s",production[i]);
    }
    for(int i=0;i<num;i++)
    {
        printf("\nGRAMMAR : : : %s",production[i]);
        non_terminal=production[i][0];
        if(non_terminal==production[i][index])
        {
            alpha=production[i][index+1];
            printf(" is left recursive.\n");
            while(production[i][index]!=0 &&
production[i][index]!='|')
                index++;
            if(production[i][index]!=0)
            {
                beta=production[i][index+1];
                printf("Grammar without left recursion:\n");
                printf("%c-
>%c%c\'",non_terminal,beta,non_terminal);
                printf("\n%c\'-
>%c%c\'|E\n",non_terminal,alpha,non_terminal);
            }
            else
                printf(" can't be reduced\n");
        }
        else
    }
}
```

```

                printf(" is not left recursive.\n");
            index=3;
        }//for
    }//main

```

### Output for Left Recursion:

```

Enter Number of Production : 4
Enter the grammar as E->E-A :
E->EA|A
A->AT|a
T->a
E->i

```

```

GRAMMAR : : : E->EA|A is left recursive.
Grammar without left recursion:
E->AE'
E'->AE'|E

```

```

GRAMMAR : : : A->AT|a is left recursive.
Grammar without left recursion:
A->aA'
A'->TA'|E

```

```

GRAMMAR : : : T->a is not left recursive.

```

```

GRAMMAR : : : E->i is not left recursive.

```

### Program for Left Factoring:

```

#include<stdio.h>
#include<string.h>
int main()
{
    char
    gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[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';
    for(j=++i,i=0;gram[j]!='\0';j++,i++)

```

```

        part2[i]=gram[j];
part2[i]='\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("\n A->%s",modifiedGram);
printf("\n X->%s\n",newGram);
}

```

Output for Left Factoring:

Enter Production : A->aE+bcD|aE+eIT

A->aE+X

X->bcD|eIT

7. Write a C program to check the validity of input string using predictive parser.

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

char prol[7][10] = { "S", "A", "A", "B", "B", "C", "C" };
char pror[7][10] = { "A", "Bb", "Cd", "aB", "@", "Cc", "@" };
char prod[7][10] = { "S->A", "A->Bb", "A->Cd", "B->aB", "B->@", "C->Cc", "C->@" };
char first[7][10] = { "abcd", "ab", "cd", "a@", "@", "c@", "@" };
char follow[7][10] = { "$", "$", "$", "a$", "b$", "c$", "d$" };
char table[5][6][10];

int numr(char c)
{
    switch (c)
    {
        case 'S':
            return 0;

        case 'A':
            return 1;

        case 'B':
            return 2;

        case 'C':
            return 3;

        case 'a':
            return 0;

        case 'b':
            return 1;

        case 'c':
            return 2;

        case 'd':
            return 3;

        case '$':
            return 4;
    }
}
```

```

    return (2);
}

int main()
{
    int i, j, k;

    for (i = 0; i < 5; i++)
        for (j = 0; j < 6; j++)
            strcpy(table[i][j], " ");

    printf("The following grammar is used for Parsing Table:\n");

    for (i = 0; i < 7; i++)
        printf("%s\n", prod[i]);

    printf("\nPredictive parsing table:\n");

    fflush(stdin);

    for (i = 0; i < 7; i++)
    {
        k = strlen(first[i]);
        for (j = 0; j < 10; j++)
            if (first[i][j] != '@')
                strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) +
1], prod[i]);
    }

    for (i = 0; i < 7; i++)
    {
        if (strlen(pror[i]) == 1)
        {
            if (pror[i][0] == '@')
            {
                k = strlen(follow[i]);
                for (j = 0; j < k; j++)
                    strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j])
+ 1], prod[i]);
            }
        }
    }

    strcpy(table[0][0], " ");

```

```

strcpy(table[0][1], "a");
strcpy(table[0][2], "b");
strcpy(table[0][3], "c");
strcpy(table[0][4], "d");
strcpy(table[0][5], "$");
strcpy(table[1][0], "S");
strcpy(table[2][0], "A");
strcpy(table[3][0], "B");
strcpy(table[4][0], "C");

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

for (i = 0; i < 5; i++)
    for (j = 0; j < 6; j++)
    {
        printf("%-10s", table[i][j]);
        if (j == 5)
            printf("\n-----
-----\n");
    }
}

```

Output:

The following grammar is used for Parsing Table:

S->A

A->Bb

A->Cd

B->aB

B->@

C->Cc

C->@

Predictive parsing table:

-----				
a	b	c	d	\$
-----				
S	S->A	S->A	S->A	S->A
-----				
A	A->Bb	A->Bb	A->Cd	A->Cd
-----				
B	B->aB	B->@	B->@	B->@
-----				
C		C->@	C->@	C->@
-----				

8 .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.

```
#include<stdio.h>

#include<string.h>

int k=0,z=0,i=0,j=0,c=0;

char a[16],ac[20],stk[15],act[10];

void check();

int main()

{

    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");

    puts("enter input string ");

    gets(a);

    c=strlen(a);

    strcpy(act,"SHIFT->");

    puts("stack \t input \t action");

    for(k=0,i=0; j<c; k++,i++,j++)

    {

        if(a[j]=='i' && a[j+1]=='d')

        {

            stk[i]=a[j];

            stk[i+1]=a[j+1];

            stk[i+2]='\0';

            a[j]=' ';

            a[j+1]=' ';

            printf("\n$%s\t%s$\t%sid",stk,a,act);

            check();

        }

    }

}
```



```

else
{
    stk[i]=a[j];
    stk[i+1]='\0';
    a[j]=' ';
    printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
    check();
}
}

void check()
{
    strcpy(ac,"REDUCE TO E");
    for(z=0; z<c; z++)
        if(stk[z]=='i' && stk[z+1]=='d')
        {
            stk[z]='E';
            stk[z+1]='\0';
            printf("\n$%s\t%s$\t%s",stk,a,ac);
            j++;
        }
    for(z=0; z<c; z++)
        if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
        {
            stk[z]='E';
            stk[z+1]='\0';
            stk[z+1]='\0';
            printf("\n$%s\t%s$\t%s",stk,a,ac);

```

```

        i=i-2;
    }
    for(z=0; z<c; z++)
    if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]=='')
    {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+1]='\0';
        printf("\n%s\t%s$\t%s",stk,a,ac);
        i=i-2;
    }
}

```

Output:

/tmp/rVDYCnu0I4.o

GRAMMAR is E->E+E

E->E\*E

E->(E)

E->id

enter input string

id+id\id+id

stack	input	action
\$id	+id\id+id\$	SHIFT->id
\$E	+id\id+id\$	REDUCE TO E
\$E+	id\id+id\$	SHIFT->symbols
\$E+id	\id+id\$	SHIFT->id
\$E+E	\id+id\$	REDUCE TO E

\$E+E\	id+id\$	SHIFT->symbols
\$E+E\id	+id\$	SHIFT->id
\$E+E\E	+id\$	REDUCE TO E
\$E+E\E+	id\$	SHIFT->symbols
\$E+E\E+id	\$	SHIFT->id
\$E+E\E+E	\$	REDUCE TO E

9. Simulate the calculator using LEX and YACC tool.

Lexfile.l

```
%option noinput nounput noyywrap
```

```
%{
```

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

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

```
#include "y.tab.h"
```

```
extern int yylval;
```

```
%}
```

```
%%
```

```
[\t] ;
```

```
[\n] return 0;
```

```
[0-9]+ { yylval = atoi(yytext);
```

```
    return num;
```

```
}
```

```
.    return yytext[0];  
%%
```

## File.y

```
%{  
#include <ctype.h>  
#include <stdio.h>  
int yylex();  
void yyerror();  
int tmp=0;  
%}
```

```
%token num
```

```
%left '+' '-'
```

```
%left '*' '/'
```

```
%left '(' ')'
```

```
%%
```

```
line :exp {printf("%d\n",$$); return 0;;}
```

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

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

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

```
    | exp '/' exp {$$=$1/$3;}
```

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

```
    | num {$$=$1;}
```

%%

```
void yyerror(){
    printf("The arithmetic expression is correct\n");
    tmp=1;
}
int main(){
    printf("Enter an arithmetic expression(can contain +,-,*,/ or parenthesis):\n");
    yyparse();
}
```

### **Output:**

To run lexfile.l the command is:

```
lex lexfile.l
```

To run file.y the command is:

```
yacc -d file.y
```

```
gcc -o output lex.yy.c y.tab.c
```

```
./output
```

Enter an arithmetic expression(can contain +,-,\*,/ or parenthesis):

a+b

The arithmetic expression is correct

10. Write a C program to generate a three address code for a given expression.

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

struct three
{
char data[10],temp[7];
}s[30];

void main()
{
char d1[7],d2[7]="t";
int i=0,j=1,len=0;
FILE *f1,*f2;

clrscr();

f1=fopen("sum.txt","r");
f2=fopen("out.txt","w");
while(fscanf(f1,"%s",s[len].data)!=EOF)
len++;

itoa(j,d1,7);
strcat(d2,d1);
strcpy(s[j].temp,d2);
strcpy(d1,"");
strcpy(d2,"t");
if(!strcmp(s[3].data,"+"))
{
```

```

fprintf(f2, "%s=%s+%s", s[j].temp, s[i+2].data, s[i+4].data);
j++;
}
else if(!strcmp(s[3].data, "-"))
{
fprintf(f2, "%s=%s-%s", s[j].temp, s[i+2].data, s[i+4].data);
j++;
}
for(i=4; i<len-2; i+=2)
{
itoa(j, d1, 7);
strcat(d2, d1);
strcpy(s[j].temp, d2);
if(!strcmp(s[i+1].data, "+"))
fprintf(f2, "\n%s=%s+%s", s[j].temp, s[j-1].temp, s[i+2].data);
else if(!strcmp(s[i+1].data, "-"))
fprintf(f2, "\n%s=%s-%s", s[j].temp, s[j-1].temp, s[i+2].data);
strcpy(d1, "");
strcpy(d2, "t");
j++;
}
fprintf(f2, "\n%s=%s", s[0].data, s[j-1].temp);
fclose(f1);
fclose(f2);
getch();
}

```

Input: sum.txt

out = in1 + in2 + in3 - in4

Output :    out.txt

t1=in1+in2

t2=t1+in3

t3=t2-in4

out=t3