Finding tokens like identifier, keywords

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

int main() {
    char input[100];
    printf("Enter a string: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = '\0'; // Remove newline character

    char* token = strtok(input, " ");
    while (token != NULL) {
        if (isalpha(token[0]) || token[0] == '_') {
            const char* keywords[] = {"if", "else", "while", "for", "int", "float", "char", "return"};
            int isKeyword = 0;
            for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {
                if (strcmp(token, keywords[i]) == 0) {
                    isKeyword = 1;
                    break;
                }
            }
            printf("'%s' is %s.\n", token, isKeyword ? "a keyword" : "an identifier");
        } else {
            printf("'%s' is not a valid token.\n", token);
        }
        token = strtok(NULL, " ");
    }

    return 0;
}
```

Left Recursion

```
#include<iostream>
#include<string>
using namespace std;

int main() {
    string ip, op1, op2, temp;
    char c;
    int n;

    cout << "Enter the Parent Non-Terminal : ";
    cin >> c;
    ip.push_back(c);
    op1 += ip + "->";
    op2 += ip + "\'->";

    cout << "Enter the number of productions : ";
    cin >> n;
    for (int i = 0; i < n; i++) {
        cout << "Enter Production " << i + 1 << " : ";
        cin >> temp;
        if (temp[0] == c) {
            temp.erase(0, 1);
            op2 += temp + c + "\'|";
        } else {
            op1 += temp + c + "\'|";
        }
    }
    op1 += "#";
    op2 += "#";
    cout << "New Productions without Left Recursion:\n";
    cout << op1 << endl;
    cout << op2 << endl;

    return 0;
}
```

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("\nGrammar Without Left Factoring : : \n");
printf(" A->%s",modifiedGram);
printf("\n X->%s\n",newGram);
}
```

FIRST

```
#include<stdio.h>
#include<ctype.h>
void FIRST(char[],char );
void addToResultSet(char[],char);
int numOfProductions;
char productionSet[10][10];
int main()
{
    int i;
    char choice;
    char c;
    char result[20];
    printf("How many number of productions ? :");
    scanf(" %d",&numOfProductions);
    for(i=0;i<numOfProductions;i++)//read production string eg: E=E+T
    {
        printf("Enter productions Number %d : ",i+1);
        scanf(" %s",productionSet[i]);
    }
    do
    {
        printf("\n Find the FIRST of  :");
        scanf(" %c",&c);
        FIRST(result,c); //Compute FIRST; Get Answer in 'result' array
        printf("\n FIRST(%c)= { ",c);
        for(i=0;result[i]!='\0';i++)
        printf(" %c ",result[i]);       //Display result
        printf("}\n");
         printf("press 'y' to continue : ");
        scanf(" %c",&choice);
    }
    while(choice=='y'||choice =='Y');
}
/*
 *Function FIRST:
 *Compute the elements in FIRST(c) and write them
 *in Result Array.
 */
```

```
void FIRST(char* Result,char c)
{
    int i,j,k;
    char subResult[20];
    int foundEpsilon;
    subResult[0]='\0';
    Result[0]='\0';
    //If X is terminal, FIRST(X) = {X}.
    if(!(isupper(c)))
    {
        addToResultSet(Result,c);
               return ;
    }
    for(i=0;i<numOfProductions;i++)
    {
        if(productionSet[i][0]==c)
        {
 if(productionSet[i][2]=='$')
 addToResultSet(Result,'$');
      else
            {
                j=2;
                while(productionSet[i][j]!='\0')
                {
                foundEpsilon=0;
                FIRST(subResult,productionSet[i][j]);
                for(k=0;subResult[k]!='\0';k++)
                    addToResultSet(Result,subResult[k]);
                 for(k=0;subResult[k]!='\0';k++)
                     if(subResult[k]=='$')
                     {
                         foundEpsilon=1;
                         break;
                     }
                 //No ε found, no need to check next element
                 if(!foundEpsilon)
                     break;
                 j++;
                }
            }
    }
}
    return ;
}
void addToResultSet(char Result[],char val)
{
    int k;
    for(k=0 ;Result[k]!='\0';k++)
        if(Result[k]==val)
            return;
    Result[k]=val;
    Result[k+1]='\0';
}
```

FOLLOW

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
int n,m=0,p,i=0,j=0;
char a[10][10],f[10];
void follow(char c);
void first(char c);
int main()
{
 int i,z;
 char c,ch;
 printf("Enter the no.of productions:");
 scanf("%d",&n);
 printf("Enter the productions(epsilon=$):\n");
 for(i=0;i<n;i++)
  scanf("%s%c",a[i],&ch);

 do
 {
  m=0;
  printf("Enter the element whose FOLLOW is to be found:");

  scanf("%c",&c);
  follow(c);
  printf("FOLLOW(%c) = { ",c);
  for(i=0;i<m;i++)
   printf("%c ",f[i]);
  printf(" }\n");
  printf("Do you want to continue(0/1)?");
  scanf("%d%c",&z,&ch);
 }
 while(z==1);
}
void follow(char c)
{

 if(a[0][0]==c)f[m++]='$';
 for(i=0;i<n;i++)
 {
  for(j=2;j<strlen(a[i]);j++)
  {
   if(a[i][j]==c)
   {
    if(a[i][j+1]!='\0')first(a[i][j+1]);

    if(a[i][j+1]=='\0'&&c!=a[i][0])
     follow(a[i][0]);

   }
  }
 }
}
void first(char c)
{
      int k;
                 if(!(isupper(c)))f[m++]=c;
                 for(k=0;k<n;k++)
                 {
                 if(a[k][0]==c)
                 {
                 if(a[k][2]=='$') follow(a[i][0]);
                 else if(islower(a[k][2]))f[m++]=a[k][2];
                 else first(a[k][2]);
                 }
                 }

}
```

SHIFT REDUCE PARSING

```
// Including Libraries
#include <bits/stdc++.h>
using namespace std;

// Global Variables
int z = 0, i = 0, j = 0, c = 0;

// Modify array size to increase
// length of string to be parsed
char a[16], ac[20], stk[15], act[10];

// This Function will check whether
// the stack contain a production rule
// which is to be Reduce.
// Rules can be S->AB, A->a, B->b
void check()
{
        // Copying string to be printed as action
        strcpy(ac,"REDUCE: ");

        // c=length of input string
        for(z = 0; z < c; z++)
        {
                // checking for producing rule B->b
                if(stk[z] == 'b')
                {
                        printf("%sB -> b", ac);
                        stk[z] = 'B';
                        stk[z + 1] = '\0';

                        //printing action
                        printf("\n$%s\t%s$\t", stk, a);
                }
        }

        for(z = 0; z < c - 1; z++)
        {
                // checking for another production A->a
                if(stk[z] == 'a')
                {
                        printf("%sA -> a", ac);
                        stk[z] = 'A';
                        stk[z + 1] = '\0';
                        printf("\n$%s\t%s$\t", stk, a);
i = i - 1;
                }
        }

        for(z = 0; z < c - 1; z++)
        {
                //checking for S->AB
                if(stk[z] == 'A' && stk[z + 1] == 'B')
                {
                        printf("%sS -> AB", ac);
                        stk[z]='S';
                        stk[z + 1]='\0';
                        printf("\n$%s\t%s$\t", stk, a);
                        i = i - 1;
                }
        }
        return ; // return to main
}

// Driver Function
int main()
{
        printf("GRAMMAR is -\nS -> AB \nA -> a \nB -> b\n");

        // a is input string
        strcpy(a,"abab");

        // strlen(a) will return the length of a to c
        c=strlen(a);

        // "SHIFT" is copied to act to be printed
        strcpy(act,"SHIFT");

        // This will print Labels (column name)
        printf("\nstack \t input \t action");


// This will print the initial
        // values of stack and input
        printf("\n$\t%s$\t", a);

        // This will Run upto length of input string
```

```
    for(i = 0; j < c; i++, j++)
    {
        // Printing action
        printf("%s", act);

        // Pushing into stack
        stk[i] = a[j];
        stk[i + 1] = '\0';

        // Moving the pointer
        a[j]=' ';

        // Printing action
        printf("\n$%s\t%s$\t", stk, a);

        // Call check function ..which will
        // check the stack whether its contain
        // any production or not
        check();
    }

    // Rechecking last time if contain
    // any valid production then it will
    // replace otherwise invalid
    check();

    // if top of the stack is S(starting symbol)
    // then it will accept the input
if(stk[0] == 'S' && stk[1] == '\0')
        printf("Accept\n");
    else //else reject
        printf("Reject\n");
}
```

PREDICTIVE PARSING

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    char fin[10][20], st[10][20], ft[20][20], fol[20][20];
    int a, i, t, b, n, j, s = 0, p;

    cout << "Enter the number of productions: ";
    cin >> n;

    cout << "Enter the productions of the grammar:\n";
    for (i = 0; i < n; i++)
        cin >> st[i];

    cout << "\nEnter the FIRST and FOLLOW of each non-terminal:";
    for (i = 0; i < n; i++)
    {
        cout << "\nFIRST[" << st[i][0] << "] : ";
        cin >> ft[i];
        cout << "FOLLOW[" << st[i][0] << "] : ";
        cin >> fol[i];
    }

    cout << "\nThe contents of the predictive parser table are:\n";
    for (i = 0; i < n; i++)
    {
        j = 3;
        while (st[i][j] != '\0')
        {
            if (st[i][j - 1] == '|' || j == 3)
            {
                for (p = 0; p <= 2; p++)
                    fin[s][p] = st[i][p];
                t = j;
                for (p = 3; st[i][j] != '|' && st[i][j] != '\0'; p++, j++)
                    fin[s][p] = st[i][j];
                fin[s][p] = '\0';

                if (st[i][t] == 'e')
                {
                    a = b = 0;
                    while (st[a++][0] != st[i][0])
                        ;
                    while (fol[i][b] != '\0')
                    {
                        cout << "M[" << st[i][0] << "," << fol[i][b]
                             << "] = " << fin[s] << "\n";
                        b++;
                    }
                }
                else if (!(st[i][t] > 64 && st[i][t] < 91))
                    cout << "M[" << st[i][0] << "," << st[i][t]
                         << "] = " << fin[s] << "\n";
                else
                {
                    a = 0;
                    while (st[a][0] != st[i][t] && a < n)
                        a++;

                    if (a < n) {
                        b = 0;
                        while (ft[a][b] != '\0')
                        {
                            cout << "M[" << st[i][0] << "," << ft[a][b]
                                 << "] = " << fin[s] << "\n";
                            b++;
                        }
                    }
                }
                s++;  // Increment index for storing entries in the parsing table.
            }
            if (st[i][j] == '|')  // If '|' encountered, move to next symbol.
                j++;
        }
    }
    return 0;
}
```

SYNTAX TREE

```
#include <iostream>
#include <stack>

using namespace std;

struct SyntaxTreeNode {
    string value;
    SyntaxTreeNode* left;
    SyntaxTreeNode* right;

    SyntaxTreeNode(string val) : value(val), left(nullptr), right(nullptr) {}
};

bool isOperator(string token) {
    return token == "+" || token == "-" || token == "*" || token == "/";
}

SyntaxTreeNode* constructSyntaxTree(string postfixExpression[], int size) {
    stack<SyntaxTreeNode*> st;

    for (int i = 0; i < size; ++i) {
        SyntaxTreeNode* newNode = new SyntaxTreeNode(postfixExpression[i]);

        if (isOperator(postfixExpression[i])) {
            SyntaxTreeNode* rightNode = st.top();
            st.pop();
            SyntaxTreeNode* leftNode = st.top();
            st.pop();
            newNode->left = leftNode;
            newNode->right = rightNode;
        }

        st.push(newNode);
    }

    return st.top();
}

void printInfix(SyntaxTreeNode* root) {
    if (root != nullptr) {
        if (isOperator(root->value)) {
            cout << "(";
        }
        printInfix(root->left);
        cout << root->value;
        printInfix(root->right);
        if (isOperator(root->value)) {
            cout << ")";
        }
    }
}

void printSyntaxTree(SyntaxTreeNode* root, string prefix, bool isLeft) {
    if (root != nullptr) {
        cout << prefix << (isLeft ? "|-- " : "\\-- ") << root->value << endl;
        printSyntaxTree(root->left, prefix + (isLeft ? "|   " : "    "), true);
        printSyntaxTree(root->right, prefix + (isLeft ? "|   " : "    "), false);
    }
}

int main() {
    string postfixExpression[] = {"3", "4", "+", "5", "*"};
    int size = sizeof(postfixExpression) / sizeof(postfixExpression[0]);

    SyntaxTreeNode* root = constructSyntaxTree(postfixExpression, size);

    cout << "Syntax Tree:" << endl;
    printSyntaxTree(root, "", true);

    cout << "\nInfix expression: ";
    printInfix(root);

    return 0;
}
```

3- ADDRESS CODE

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

const vector<vector<char>> precedence = {
    {'/', '1'},
    {'*', '1'},
    {'+', '2'},
    {'-', '2'}
};

int precedenceOf(char token) {
    for (size_t i = 0; i < precedence.size(); i++) {
        if (token == precedence[i][0]) {
            return precedence[i][1] - '0';
        }
    }
    return -1;
}

int main() {
    int i, j, opc = 0;
    char token;
    vector<vector<string>> operators(10, vector<string>(2));
    string expr, temp;
    bool processed[expr.length()] = {false};

    cout << "\nEnter an expression for calculating Address codes: ";
    getline(cin, expr);

    for (i = 0; i < expr.length(); i++) {
        processed[i] = false;
    }

    for (i = 0; i < expr.length(); i++) {
        token = expr[i];
        for (j = 0; j < precedence.size(); j++) {
```

```
        if (token == precedence[j][0]) {
            operators[opc][0] = token;
            operators[opc][1] = to_string(i);
            opc++;
            break;
        }
    }
}

cout << "\nOperators: \nOperators \tLocation number\n";
for (i = 0; i < opc; i++) {
    cout << operators[i][0] << "\t\t" << operators[i][1] << endl;
}

for (i = opc - 1; i >= 0; i--) {
    for (j = 0; j < i; j++) {
        if (precedenceOf(operators[j][0][0]) > precedenceOf(operators[j + 1][0][0])) {
            temp = operators[j][0];
            operators[j][0] = operators[j + 1][0];
            operators[j + 1][0] = temp;
            temp = operators[j][1];
            operators[j][1] = operators[j + 1][1];
            operators[j + 1][1] = temp;
        }
    }
}

cout << "\nOperators sorted in their precedence: \nOperators \tLocation number \n";
for (i = 0; i < opc; i++) {
    cout << operators[i][0] << "\t\t" << operators[i][1] << endl;
}

cout << endl;
for (i = 0; i < opc; i++) {
    j = stoi(operators[i][1]);
    string op1 = "", op2 = "";
    if (processed[j - 1]) {
        if (precedenceOf(operators[i - 1][0][0]) == precedenceOf(operators[i][0][0])) {
            op1 = "t" + to_string(i);
        } else {
                for (int x = 0; x < opc; x++) {
                    if ((j - 2) == stoi(operators[x][1])) {
                        op1 = "t" + to_string(x + 1);
                    }
                }
            }
        } else {
            op1 = expr[j - 1];
        }
        if (processed[j + 1]) {
            for (int x = 0; x < opc; x++) {
                if ((j + 2) == stoi(operators[x][1])) {
                    op2 = "t" + to_string(x + 1);
                }
            }
        } else {
            op2 = expr[j + 1];
        }
        cout << "t" << (i + 1) << " = " << op1 << operators[i][0] << op2 << endl;
        processed[j] = processed[j - 1] = processed[j + 1] = true;
    }

    return 0;
}
```

```
    }
```

STRING COMPARISON

```
import java.util.Scanner;

public class StringComparisonExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter the first string:");
        String str1 = scanner.nextLine();

        System.out.println("Enter the second string:");
        String str2 = scanner.nextLine();

        // Using equalsIgnoreCase() to compare strings
        if (str1.equalsIgnoreCase(str2)) {
            System.out.println("The strings are equal (ignoring case).");
        } else {
            System.out.println("The strings are not equal (ignoring case).");
        }

        // Using equals() to compare strings
        if (str1.equals(str2)) {
            System.out.println("The strings are equal (case-sensitive).");
        } else {
            System.out.println("The strings are not equal (case-sensitive).");
        }

        // Using compareTo() to compare strings
        int result = str1.compareTo(str2);
        if (result == 0) {
            System.out.println("The strings are equal.");
        } else if (result < 0) {
            System.out.println("str1 comes before str2 in lexicographic order.");
        } else {
            System.out.println("str1 comes after str2 in lexicographic order.");
        }

        scanner.close();
    }
}
```