Compiler Construction
Practical Exam Questions with Solutions

1. Removal of comments, extra blank spaces and Token counting.
Ans)

```
//  Removal of comments, extra blank spaces and Token counting
// Assumption that the language does not support strings.
#include <iostream>
#include <fstream>
#include <regex>
#include <process.h>
using namespace std;
int main(){
    ifstream fin;
    ofstream fout;
    string text, remove;
    int tokens = 0;
    // Resultant File
    fout.open("tokens.txt",ios::trunc|ios::out);
    if(!fout){
        cout<<"The file could not be created."<<endl;
        exit(0);
    }
    // Input File
    fin.open("code.txt",ios::in);
    while(!fin.eof()){
        fin>>text;
        // removal of preprocessor directives
        if(text[0]=='#'){
            getline(fin,remove);
            continue;
        }
        // removal of comments
        if((text[0]=='/' && text[1]=='*') ||
regex_match(text,regex("(//)(.*)"))){
            if(text[1]=='*'){
                getline(fin,remove,'/');
            }
            else{
                getline(fin,remove);
            }
            continue;
        }
        // removal of extra blank spaces
        if(text==""){
            continue;
        }
        fout<<text<<" ";
```

```
        // token counting
        tokens++;
        text="";
    }
    fin.close();
    fout.close();
    cout<<"No of tokens present in the file are: "<<tokens<<endl;
}
```

OR

```
//  Removal of comments, extra blank spaces and Token counting
// Assumption – The language supports strings.
#include <iostream>
#include <fstream>
#include <regex>
#include <process.h>
using namespace std;
int main(){
    ifstream fin;
    ofstream fout;
    string text, remove;
    int tokens = 0, count = 0;
    bool flag = false;
    // Resultant File
    fout.open("tokens.txt",ios::trunc|ios::out);
    if(!fout){
        cout<<"The file could not be created."<<endl;
        exit(0);
    }
    // Input File
    fin.open("code.txt",ios::in);
    while(!fin.eof()){
        fin>>text;
        // removal of preprocessor directives
        if(text[0]=='#'){
            getline(fin,remove);
            continue;
        }
        // removal of comments
        if((text[0]=='/' && text[1]=='*') ||
regex_match(text,regex("(//)(.*)"))){
            if(text[1]=='*'){
                getline(fin,remove,'/');
            }
            else{
                getline(fin,remove);
            }
            continue;
        }
```

```
        // removal of extra blank spaces
        if(text==""){
            continue;
        }
        fout<<text<<" ";
        // token counting
        tokens++;
        int n = text.size()-1;
        if((text[0]=='\"' || text[0]=='\'')||(text[n]=='\"' ||
text[n]=='\'')){
            if(flag==false){
                flag=true;
            }
            else{
                flag=false;
            }
        }
        if(flag==true){
            count++;
        }
        //cout<<tokens<<" "<<text<<endl;
        text="";
    }
    fin.close();
    fout.close();
    cout<<"No of tokens present in the file are: "<<tokens-count<<endl;
}
```

2. To token categorisation.
Ans)

```
// To token categorisation
// Assumption - The language does not support strings
#include <iostream>
#include <fstream>
#include <regex>
#include <string>
#include <process.h>
using namespace std;
void categorize(string token, int &key, int &relop, int &arithop, int &obj,
int &iden){
    // Keywords
    string keywords[32] =
{"auto","break","case","char","const","continue","default","do",
                        "double","else","enum","extern","for","float","goto","
if",
                        "int","long","register","return","short","signed","siz
eof","static",
```

```
                        "struct","switch","typedef","union","unsigned","void",
"volatile","while"};
    // predefined identifier
    string objects[7] =
{"cout","cin","endl","main","using","namespace","std"};
    // relational operators
    string relOps[6] = {">","<",">=","<=","==","!="};
    // arithmatic operators
    string arithOps[6] = {"+","-","*","/","%","="};
    for(int i=0;i<32;i++){
        if(token.compare(keywords[i])==0){
            cout<<token<<" is a keyword."<<endl;
            key++;
            return;
        }
    }
    for(int i=0;i<7;i++){
        if(token.compare(objects[i])==0){
            cout<<token<<" is a predefined identifier."<<endl;
            obj++;
            return;
        }
    }
    for(int i=0;i<6;i++){
        if(token.compare(relOps[i])==0){
            cout<<token<<" is a relational operator."<<endl;
            relop++;
            return;
        }
    }
    for(int i=0;i<6;i++){
        if(token.compare(arithOps[i])==0){
            cout<<token<<" is an arithmatic operator."<<endl;
            arithop++;
            return;
        }
    }
    if(token[0]=='_'||(token[0]>='A' && token[0]<='Z') || (token[0]>='a' &&
token[0]<='z')){
        cout<<token<<" is an identifier."<<endl;
        iden++;
        return;
    }
}
int main(){
    ifstream fin;
    ofstream fout;
    string text, remove;
```

```
    int tokens = 0, keywords = 0, identifiers = 0, relop = 0, arithop = 0,
objects = 0;
    // Resultant File
    fout.open("tokens.txt",ios::trunc|ios::out);
    if(!fout){
        cout<<"The file could not be created."<<endl;
        exit(0);
    }
    // Input File
    fin.open("code.txt",ios::in);
    while(!fin.eof()){
        fin>>text;
        // removal of preprocessor directives
        if(text[0]=='#'){
            getline(fin,remove);
            continue;
        }
        // removal of comments
        if((text[0]=='/' && text[1]=='*') ||
regex_match(text,regex("(//)(.*)"))){
            if(text[1]=='*'){
                getline(fin,remove,'/');
            }
            else{
                getline(fin,remove);
            }
            continue;
        }
        // removal of extra blank spaces
        if(text==""){
            continue;
        }
        fout<<text<<" ";
        // token counting
        tokens++;
        // categorization of tokens
        categorize(text,keywords,relop,arithop,objects,identifiers);
        text="";
    }
    fin.close();
    fout.close();
    //cout<<"No of tokens present in the file are: "<<tokens<<endl;
    cout<<"No of keywords present in the file are: "<<keywords<<endl;
    cout<<"No of identifiers present in the file are: "<<identifiers<<endl;
    cout<<"No of predefined identifiers present in the file are:
"<<objects<<endl;
    cout<<"No of relational operators present in the file are: "<<relop<<endl;
```

```
    cout<<"No of arithmatic operators present in the file are:
"<<arithop<<endl;
}
```

3. Infix to postfix.
Ans)

```
//  Infix to postfix
//a+b*(c^d-e)^(f+g*h)-i
#include <iostream>
#include <stack>
#include <string>
using namespace std;
bool checkPths(string pths){
    stack<char> s;
    //bool b = true;
    if(pths[0]==']' || pths[0]==')' || pths[0]=='}'){
        return false;
    }
    else{
        for(int i=0;i<pths.size();i++){
            if(pths[i]=='[' || pths[i]=='(' || pths[i]=='{'){
                s.push(pths[i]);
            }
            else if(pths[i]==']'){
                if(s.top()=='['){
                    s.pop();
                }
                else{
                    return false;
                }
            }
            else if(pths[i]=='}'){
                if(s.top()=='{'){
                    s.pop();
                }
                else{
                    return false;
                }
            }
            else if(pths[i]==')'){
                if(s.top()=='('){
                    s.pop();
                }
                else{
                    return false;
                }
            }
            else{
```

```
                continue;
            }
        }
        if(s.size()==0){
            return true;
        }
        else{
            return false;
        }
    }
}
bool checkOps(string expr){
    bool b = true;
    for(int i=0;i<expr.length()-2;i+=2){
        if(((expr[i]>='a' && expr[i]<='z')||(expr[i]>='A' && expr[i]<='Z')||
            (expr[i]>='0' && expr[i]<='9')) &&
            (expr[i+1]=='+'||expr[i+1]=='-
'||expr[i+1]=='*'||expr[i+1]=='/'||expr[i+1]=='^') &&
            ((expr[i+2]>='a' && expr[i+2]<='z')||
            (expr[i+2]>='A' && expr[i+2]<='Z')||(expr[i+2]>='0' &&
expr[i+2]<='9'))){
                b = true;
        }
        else{
            b=false;
            break;
        }
    }
    return b;
}
bool checkExpr(string inExp){
    string pths="", expr="";
    for(int i=0;i<inExp.size();i++){
        if(inExp[i]=='(' || inExp[i]==')' || inExp[i]=='[' ||
            inExp[i]==']' || inExp[i]=='}'|| inExp[i]=='{'){
            pths += inExp[i];
        }
        else{
            expr += inExp[i];
        }
    }
    // Checking of Parenthesis and Expression Ordering
    if(checkPths(pths) && checkOps(expr)){
        return true;
    }
    else{
        return false;
    }
```

```
}
string infixToPostfix(string inExp){
    stack <char> s;
    string expr="";
    for(int i=0;i<inExp.length();i++){
        //Parenthesis
        if(inExp[i]=='('||inExp[i]=='['||inExp[i]=='{'){
            s.push(inExp[i]);
        }
        else if(inExp[i]==')'){
            while(!s.empty() && s.top()!='('){
                expr+=s.top();
                s.pop();
            }
            s.pop();
        }
        else if(inExp[i]==']'){
            while(!s.empty() && s.top()!='['){
                expr+=s.top();
                s.pop();
            }
            s.pop();
        }
        else if(inExp[i]=='}'){
            while(!s.empty() && s.top()!='{'){
                expr+=s.top();
                s.pop();
            }
            s.pop();
        }
        //Operators
        // '^' operator
        else if(inExp[i]=='^'){
            if(s.empty()){
                s.push(inExp[i]);
            }
            else
            {
                char a = s.top();
                if(a=='^'){
                    expr += a;
                    s.pop();
                    while(!s.empty()){
                        char b = s.top();
                        if(b=='^'){
                            expr += b;
                            s.pop();
                        }
```

```
                else{
                    break;
                }
            }
        s.push(inExp[i]);
        }
        else{
            s.push(inExp[i]);
        }
    }
}
// '*' operator
else if(inExp[i]=='*'){
    if(s.empty()){
        s.push(inExp[i]);
    }
    else
    {
        char a = s.top();
        if(a == '^'||a == '*'||a=='/'){
            while(!s.empty()){
                char b = s.top();
                if(b=='^'||b=='*'||b=='/'){
                    expr += b;
                    s.pop();
                }
                else{
                    break;
                }
            }
        s.push(inExp[i]);
        }
        else{
            s.push(inExp[i]);
        }
    }
}
// '/' operator
else if(inExp[i]=='/'){
    if(s.empty()){
        s.push(inExp[i]);
    }
    else
    {
        char a = s.top();
        if(a == '^'||a == '*'||a=='/'){
            while(!s.empty()){
                char b = s.top();
```

```
                if(b=='^'||b=='*'||b=='/'){
                    expr += b;
                    s.pop();
                }
                else{
                    break;
                }
            }
        s.push(inExp[i]);
        }
        else{
            s.push(inExp[i]);
        }
    }
}
// '+' operator
else if(inExp[i]=='+'){
    if(s.empty()){
        s.push(inExp[i]);
    }
    else
    {
        char a = s.top();
        if(a=='^'||a=='*'||a=='/'||a=='+'||a=='-'){
            while(!s.empty()){
                char b = s.top();
                if(b=='^'||b=='*'||b=='/'||b=='+'||b=='-'){
                    expr += b;
                    s.pop();
                }
                else{
                    break;
                }
            }
        s.push(inExp[i]);
        }
        else{
            s.push(inExp[i]);
        }
    }
}
// '-' operator
else if(inExp[i]=='-'){
    if(s.empty()){
        s.push(inExp[i]);
    }
    else
    {
```

```
                char a = s.top();
                if(a=='^'||a=='*'||a=='/'||a=='+'||a=='-'){
                    while(!s.empty()){
                        char b = s.top();
                        if(b=='^'||b=='*'||b=='/'||b=='+'||b=='-'){
                            expr += b;
                            s.pop();
                        }
                        else{
                            break;
                        }
                    }
                s.push(inExp[i]);
                }
                else{
                    s.push(inExp[i]);
                }
            }
        }
        else{
            expr+=inExp[i];
        }
    }
    while(!s.empty()){
        expr += s.top();
        s.pop();
    }
    return expr;
}
int main()
{
    string inExp;
    while(1){
        cout<<"Enter the infix expression: ";
        cin>>inExp;
        // Checking of Infix Expression
        if(checkExpr(inExp)){
            break;
        }
        else{
            cout<<"Enter the correct infix expression."<<endl;
        }
    }
    //Conversion of Infix to Postfix
    string postExp = infixToPostfix(inExp);
    cout<<"The postfix expression is: "<<postExp<<endl;
    return 0;
}
```

OR

```
//  Infix to postfix
//a+b*(c^d-e)^(f+g*h)-i
#include <iostream>
#include <stack>
#include <string>
using namespace std;
bool checkPths(string pths){
    stack<char> s;
    //bool b = true;
    if(pths[0]==']' || pths[0]==')' || pths[0]=='}'){
        return false;
    }
    else{
        for(int i=0;i<pths.size();i++){
            if(pths[i]=='[' || pths[i]=='(' || pths[i]=='{'){
                s.push(pths[i]);
            }
            else if(pths[i]==']'){
                if(s.top()=='['){
                    s.pop();
                }
                else{
                    return false;
                }
            }
            else if(pths[i]=='}'){
                if(s.top()=='{'){
                    s.pop();
                }
                else{
                    return false;
                }
            }
            else if(pths[i]==')'){
                if(s.top()=='('){
                    s.pop();
                }
                else{
                    return false;
                }
            }
            else{
                continue;
            }
        }
        if(s.size()==0){
            return true;
```

```
        }
        else{
            return false;
        }
    }
}
bool checkOps(string expr){
    bool b = true;
    for(int i=0;i<expr.length()-2;i+=2){
        if(((expr[i]>='a' && expr[i]<='z')||(expr[i]>='A' && expr[i]<='Z')||
            (expr[i]>='0' && expr[i]<='9')) &&
            (expr[i+1]=='+'||expr[i+1]=='-
'||expr[i+1]=='*'||expr[i+1]=='/'||expr[i+1]=='^') &&
            ((expr[i+2]>='a' && expr[i+2]<='z')||
            (expr[i+2]>='A' && expr[i+2]<='Z')||(expr[i+2]>='0' &&
expr[i+2]<='9'))){
                b = true;
        }
        else{
            b=false;
            break;
        }
    }
    return b;
}
bool checkExpr(string inExp){
    string pths="", expr="";
    for(int i=0;i<inExp.size();i++){
        if(inExp[i]=='(' || inExp[i]==')' || inExp[i]=='[' ||
            inExp[i]==']' || inExp[i]=='}'|| inExp[i]=='{'){
            pths += inExp[i];
        }
        else{
            expr += inExp[i];
        }
    }
    // Checking of Parenthesis and Expression Ordering
    if(checkPths(pths) && checkOps(expr)){
        return true;
    }
    else{
        return false;
    }
}
int prec(char c){
    if(c=='^'){
        return 3;
    }
```

```
    else if(c=='*'||c=='/'){
        return 2;
    }
    else if(c=='+'||c=='-'){
        return 1;
    }
    else{
        return -1;
    }
}
string infixToPostfix(string inExp){
    stack <char> s;
    string expr="";
    for(int i=0;i<inExp.length();i++){
        char c = inExp[i];
        //Operands
        if((c>='a' && c<='z')||(c>='A' && c<='Z')||
                (c>='0' && c<='9')) {
            expr+=c;
        }
        //Parenthesis
        else if(c=='('||c=='['||c=='{'){
            s.push(c);
        }
        else if(c==')'){
            while(!s.empty() && s.top()!='('){
                expr+=s.top();
                s.pop();
            }
            s.pop();
        }
        else if(c==']'){
            while(!s.empty() && s.top()!='['){
                expr+=s.top();
                s.pop();
            }
            s.pop();
        }
        else if(c=='}'){
            while(!s.empty() && s.top()!='{'){
                expr+=s.top();
                s.pop();
            }
            s.pop();
        }
        //Operators
        else{
            while(!s.empty() && prec(inExp[i])<=prec(s.top())){
```

```
                if(c=='^' && s.top()=='^'){
                    break;
                }
                else{
                    expr+=s.top();
                    s.pop();
                }
            }
            s.push(inExp[i]);
        }
    }
    while(!s.empty()){
        expr += s.top();
        s.pop();
    }
    return expr;
}
int main()
{
    string inExp;
    while(1){
        cout<<"Enter the infix expression: ";
        cin>>inExp;
        // Checking of Infix Expression
        if(checkExpr(inExp)){
            break;
        }
        else{
            cout<<"Enter the correct infix expression."<<endl;
        }
    }
    //Conversion of Infix to Postfix
    string postExp = infixToPostfix(inExp);
    cout<<"The postfix expression is: "<<postExp<<endl;
    return 0;
}
```

4. Postfix to infix.
Ans)

```
// Postfix to infix
#include <iostream>
#include <string>
#include <stack>
using namespace std;
bool is_operand(char ch){
    if((ch>='a' && ch<='z')||(ch>='A' && ch<='Z')||(ch>='0' && ch<='9')){
        return true;
    }
```

```
    return false;
}
string postfixToInfix(string inp){
    string out;
    stack<string> s;
    for(int i=0;i<inp.length();i++){
        if(is_operand(inp[i])){
            string op(1,inp[i]);
            s.push(op);
        }
        else{
            string op1 = s.top();
            s.pop();
            string op2 = s.top();
            s.pop();
            s.push("("+op2+inp[i]+op1+")");
        }
    }
    return s.top();
}
int main(){
    string postExp;
    cout<<"Enter the postfix expression: ";
    cin>>postExp;
    string inExp = postfixToInfix(postExp);
    cout<<"The infix expression is: "<<inExp<<endl;
    return 0;
}
```

5. Check whether a string belongs to E-> E+ E / E*E / id
Ans)

```
// check whether a string belongs to E-> E+ E / E*E / id
#include <iostream>
#include <string>
#include <stack>
#include <vector>
#include <algorithm>
#include <climits>
#include <process.h>
using namespace std;
vector <string> prod = {"E->TR","R->+TR","R->#","T->FY","Y->*FY","Y->#","F-
>i"};
vector <char> term = {'+','*','i'};
vector <char> tabRow = {'+','*','i','$'};
vector <char> nonterm = {'E','R','T','Y','F'};
int parsingTable[5][4] = {{INT_MAX,INT_MAX,0,INT_MAX},
                          {1,INT_MAX,INT_MAX,2},
                          {INT_MAX,INT_MAX,3,INT_MAX},
```

```
                            {5,4,INT_MAX,5},
                            {INT_MAX,INT_MAX,6,INT_MAX}};
bool check(char ch){
    for(int i=0;i<term.size();i++){
        if(term.at(i)==ch){
            return true;
        }
    }
    return false;
}
int main(){
    string input;
    cout<<"Enter the string: ";
    cin>>input;
    for(int i=0;i<input.length();i++){
        //cout<<input[i]<<endl;
        if(!check(input[i])){
            //cout<<"Error"<<endl;
            cout<<"The given string does not belong to the grammar";
            exit(0);
        }
    }
    stack <char> s;
    input = input +"$";
    s.push('$');
    s.push(nonterm.at(0));
    int itr=0;
    //flag = 0
    while(s.top()!='$'){
        char a = input[itr];
        char X = s.top();
        int t = find(tabRow.begin(),tabRow.end(),a) - tabRow.begin();
        int nt = find(nonterm.begin(),nonterm.end(),X) - nonterm.begin();
        if(!isupper(X) && X==a){
            s.pop();
            itr++;
        }
        else if(!isupper(X) && X!=a){
            //flag = 1;
            break;
        }
        else if(parsingTable[nt][t] == INT_MAX){
            //flag = 1;
            break;
        }
        else{
            int ind = parsingTable[nt][t];
            string pro = prod[ind].substr(3);
```

```
                s.pop();
                for(int i=pro.length()-1;i>=0;i--){
                    if(pro[i]!='#'){
                        s.push(pro[i]);
                    }
                }
            }
        }
        if(s.size()==1){
            cout<<"The given string belongs to the grammar";
        }
        else{
            cout<<"The given string does not belong to the grammar";
        }
}
```