University of Engineering and Technology, Lahore

Department of Computer Science

Compiler Construction Lab 1 Solution

Name: Muhammad Zubair

Roll No: 2022-CS-20

Course Instructor: Ms.Aliya Farooq

Date of Submission: September 16, 2025

Contents

1 Objective **2**

2 Requirements **2**

3 Program Code **2**

4 Compilation and Execution **5**

5 Sample Output **5**

5.1 Infix Example 5

5.2 Postfix Example 6

5.3 Prefix Example 6

6 Conclusion **6**

1 Objective

The objective of this lab is to design and implement a C++ program that accepts an arithmetic expression in **infix**, **prefix**, or **postfix** form, detects the type of the expression, tokenizes it into numbers, operators, and brackets, and evaluates it step by step, showing each token and applied operator.

2 Requirements

- Automatically detect whether the expression is infix, prefix, or postfix.
- Tokenize numbers (integers and floats), operators, and brackets.
- Print each token as it is processed.
- Apply operators step by step and show left operand, right operand, and result.
- Handle parentheses and brackets properly.
- Produce the final result after evaluation.

3 Program Code

The following C++ program implements the solution:

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

// Utility: check if string is operator
bool isOp(const string &s){
    return s=="+" || s=="-" || s=="*" || s=="/";
}

// precedence for shunting yard
int prec(const string &op){
    if(op=="+"||op=="-") return 1;
    if(op=="*"||op=="/") return 2;
    return 0;
}

// map closing to opening bracket
char matchingOpen(char c){
    if(c==')') return '(';
    if(c==']') return '[';
    if(c=='}') return '{';
    return 0;
}

// Tokenize infix expressions
vector<string> tokenizeInfix(const string &s){
    vector<string> tokens;
    int n=s.size();
    for(int i=0;i<n;){
        if(isspace((unsigned char)s[i])) { ++i; continue; }
        if(isdigit((unsigned char)s[i]) || (s[i]=='.' && i+1<n && isdigit((unsigned char)s[i+1]))){
            int j=i; bool dot=false;
            while(j<n && (isdigit((unsigned char)s[j]) || s[j]=='.')){
                if(s[j]=='.'){ if(dot) break; dot=true; }
                ++j;
            }
            tokens.push_back(s.substr(i,j-i));
            i=j;
        } else {
            char c = s[i];
            if(c=='+'||c=='-'||c=='*'||c=='/'||c=='('||c==')'||c=='['||c==']'||c=='{'||c=='}'){
                string t(1,c);
                tokens.push_back(t);
                ++i;
            } else {
                int j=i;
                while(j<n && !isspace((unsigned char)s[j]) && string("+-*/()[]{}").find(s[j])==string::npos) ++j;
                tokens.push_back(s.substr(i,j-i));
                i=j;
            }
        }
    }
    return tokens;
}

// Tokenize prefix/postfix (space separated)
vector<string> tokenizeSpaceSeparated(const string &s){
    vector<string> tokens; istringstream iss(s); string t;
    while(iss >> t) tokens.push_back(t);
    return tokens;
}

// Infix     Postfix (Shunting-yard algorithm)
vector<string> infixToPostfix(const vector<string> &tokens){
    vector<string> output, st;
    for(const string &tok : tokens){
        if(tok.empty()) continue;
        if(isdigit(tok[0]) || tok[0]=='.'){ output.push_back(tok); continue; }
        if(tok=="("||tok=="["||tok=="{"){ st.push_back(tok); }
        else if(tok==")"||tok=="]"||tok=="}"){
            char open = matchingOpen(tok[0]);
            while(!st.empty() && st.back()[0]!=open){ output.push_back(st.back()); st.pop_back(); }
            if(!st.empty()) st.pop_back();
        }
        else if(isOp(tok)){
            while(!st.empty() && isOp(st.back()) && prec(st.back())>=prec(tok)){
                output.push_back(st.back()); st.pop_back();
            }
            st.push_back(tok);
        } else output.push_back(tok);
    }
    while(!st.empty()){ output.push_back(st.back()); st.pop_back(); }
    return output;
}

// Evaluate postfix with steps
pair<double,bool> evalPostfixWithSteps(const vector<string> &tokens){
    vector<double> st;
    for(const string &tok : tokens){
        cout << "Token: " << tok << "\n";
        if(isOp(tok)){
            if(st.size()<2){ cerr<<"Error: not enough operands\n"; return {0,false}; }
            double r=st.back(); st.pop_back();
            double l=st.back(); st.pop_back();
            double res=0;
            if(tok=="+") res=l+r; else if(tok=="-") res=l-r;
            else if(tok=="*") res=l*r; else if(tok=="/"){ if(r==0){cerr<<"Error: div by 0\n"; return{0,false};} res=l/r; }
            cout << "Applied: " << tok << " | Left="<<l<<" Right="<<r<<" Result="<<res<<"\n";
            st.push_back(res);
        } else {
            try{ st.push_back(stod(tok)); }
            catch(...){ cerr<<"Warning: ’"<<tok<<"’ treated as 0\n"; st.push_back(0); }
        }
    }
    return {st.back(),true};
}

// Evaluate prefix with steps
pair<double,bool> evalPrefixWithSteps(const vector<string> &tokens){
    vector<double> st;
    for(auto it=tokens.rbegin(); it!=tokens.rend(); ++it){
        cout<<"Token: "<<*it<<"\n";
        if(isOp(*it)){
            if(st.size()<2){ cerr<<"Error\n"; return{0,false}; }
            double l=st.back(); st.pop_back();
            double r=st.back(); st.pop_back();
            double res=0;
            if(*it=="+") res=l+r; else if(*it=="-") res=l-r;
            else if(*it=="*") res=l*r; else if(*it=="/"){ if(r==0){cerr<<"Div0\n"; return{0,false};} res=l/r; }
            cout<<"Applied: "<<*it<<" | Left="<<l<<" Right="<<r<<" Result="<<res<<"\n";
            st.push_back(res);
        } else { try{ st.push_back(stod(*it)); } catch(...){ st.push_back(0); } }
    }
    return {st.back(),true};
}

// Detect form
string detectForm(const string &s){
    auto toks=tokenizeSpaceSeparated(s);
    if(!toks.empty()){ if(isOp(toks.front())) return "prefix"; if(isOp(toks.back())) return "postfix"; }
    return "infix";
}
```

```

int main(int argc,char**argv){
    if(argc<2){ cerr<<"Usage: "<<argv[0]<<" \"<expression>\"\n"; return 1; }
    string expr; for(int i=1;i<argc;i++){ if(i>1) expr+=" "; expr+=argv[i]; }
    cout<<"Expression: "<<expr<<"\n";
    string form=detectForm(expr);
    if(form=="infix"){
        cout<<"Detected: INFIX\n";
        auto toks=tokenizeInfix(expr);
        for(auto&t:toks) cout<<"Token: "<<t<<"\n";
        auto post=infixToPostfix(toks);
        cout<<"\n-- Evaluating Postfix --\n";
        auto res=evalPostfixWithSteps(post);
        cout<<"Result: "<<res.first<<"\n";
    } else if(form=="postfix"){
        cout<<"Detected: POSTFIX\n";
        auto toks=tokenizeSpaceSeparated(expr);
        auto res=evalPostfixWithSteps(toks);
        cout<<"Result: "<<res.first<<"\n";
    } else {
        cout<<"Detected: PREFIX\n";
        auto toks=tokenizeSpaceSeparated(expr);
        auto res=evalPrefixWithSteps(toks);
        cout<<"Result: "<<res.first<<"\n";
    }
}
```

Listing 1: Compiler Construction Lab 1 Solution in C++

4 Compilation and Execution

The program requires a C++17 compatible compiler (e.g., g++). To compile and run:

```
g++ -std=c++17 -O2 -o expr_eval expr_eval.cpp
./expr_eval "3 * (2 + 4) - [5 - 2]"
./expr_eval "2 3 4 + * 5 2 - -"
./expr_eval "- * 3 + 2 4 - 5 2"
```

5 Sample Output

5.1 Infix Example

```
Expression: 3 * (2 + 4) - [5 - 2]
Detected: INFIX
Token: 3
Token: *
Token: (
Token: 2
Token: +
Token: 4
Token: )
```

```
Token: -
Token: [
Token: 5
Token: -
Token: 2
Token: ]
-- Evaluating Postfix --
Token: 3
Token: 2
Token: 4
Token: +
Applied: + | Left=2 Right=4 Result=6
...
Result: 15
```

5.2 Postfix Example

```
Expression: 2 3 4 + * 5 2 - -
Detected: POSTFIX
Token: 2
Token: 3
Token: 4
Token: +
Applied: + | Left=3 Right=4 Result=7
Token: *
Applied: * | Left=2 Right=7 Result=14
...
Result: 11
```

5.3 Prefix Example

```
Expression: - * 3 + 2 4 - 5 2
Detected: PREFIX
Token: -
Token: *
Token: 3
Token: +
Token: 2
Token: 4
...
Result: 13
```

6 Conclusion

This program successfully evaluates infix, prefix, and postfix expressions. It prints each token, applies operators step by step, handles brackets, and displays the final result. The implementation fulfills all requirements of Lab 1 in Compiler Construction.