DEPARTMENT OF COMPUTER ENGINEERING

Semester	T.E. Semester VI– SPCC
Subject	SPCC
Subject Professor In-charge	Prof. Pankaj Vanvari
Assisting Teachers	Prof. Pankaj Vanvari
Laboratory	M310B

Student Name	Pranav Redij
Roll Number	21102A0003
TE Division	A

Title:

Operator Precedence Parser

Approach:

1. **Tokenization**: The function starts by extracting tokens from the input and storing them in a vector of vectors called **Tokensed**. Each token is represented as a pair containing its type and value.

2. **Operator Precedence Table (OPT)**: The function defines an operator precedence table as a map of pairs of tokens to their precedence relationship (**<**, **>**, or **=**).

3. **Grammar Definition**: Your grammar rules are defined as a vector of vector of vectors named **grammar**. Each production rule consists of a left-hand side non-terminal followed by its corresponding right-hand side symbols.

4. **Parsing Loop**: The function iterates over the input tokens and uses a stack to perform the parsing. It compares the precedence of the top of the stack with the incoming token based on the operator precedence table.

5. **Parsing Actions**:

    - If the top of the stack has lower precedence than the incoming token (**<** in OPT), the incoming token is pushed onto the stack.

    - If the top of the stack has higher precedence than the incoming token (**>** in OPT), it reduces the stack contents based on the grammar rules until the precedence condition is met.

- If the top of the stack and the incoming token have equal precedence (= in OPT), it pops the stack.

6. **Grammar Validation**: After each reduction step, the function validates whether the reduced symbols form a valid production according to the grammar rules.

7. **Parsing Completion**: Once the entire input has been processed, the function checks if the remaining stack contains only the start symbol and the end-of-file marker. If so, it indicates successful parsing; otherwise, it indicates a syntax error.

8. **Special Case Handling**: There's a special case handling at the end of parsing to handle cases where the last symbol on the stack is **E** and the incoming token is **$**, ensuring correct parsing completion.

9. **Output**: The function provides verbose output during the parsing process, including the current stack contents, remaining input, and production reductions.

10. **Return Value**: The function returns **true** if the input string follows the grammar and **false** otherwise.

Implementation:

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

int readfile(string &fileName, vector<string> &input)
{
    char ch;
    fstream fp;
    fp.open(fileName.c_str(), std::fstream::in);

    if (!fp)
    {
        cerr << "Error opening the file: " << fileName << endl;
        return 1; // Return an error code
    }

    string word;
    while (fp >> noskipws >> ch)
    {
        if (ch == '\n')
        {
```

```
            input.push_back(word);
            input.push_back(";");
            word = "";
        }
        else if (ch == ' ')
        {
            input.push_back(word);
            word = "";
        }
        else
        {
            word += ch;
        }
    }

    input.push_back(word);

    fp.close();
    return 0; // Return success code
}

void print_vector_2D(vector<vector<string>> &input)
{
    for (int i = 0; i < input.size(); i++)
    {
        cout << input[i][0] << " "
             << "*->" << input[i][1] << " ";
        cout << endl;
    }
    cout << endl;
}

void print_vector(vector<string> &input)
{
    for (int i = 0; i < input.size(); i++)
    {
        cout << input[i] << " ";
    }
    cout << endl;
}

void Tokenization(vector<string> &input, vector<vector<string>> &Tokensed,
map<string, string> &keywords, map<string, int> &intcp, map<string, float>
&floatcp, map<string, string> &idp)
```

```
{
    int idc = 0;
    int intcc = 0;
    int floatcc = 0;

    for (int i = 0; i < input.size(); i++)
    {
        if (input[i] == ";")
            continue;

        if (keywords.find(input[i]) != keywords.end())
        {

            Tokensed.push_back({keywords[input[i]], "NA"});
        }
        else
        {
            // if the value in not in keyword db it can be eithre identifier or
constant

            string curr = input[i];
            char first_of_curr = curr[0]; // foc
            int val_of_foc = first_of_curr - '0';
            // cout << val_of_foc << endl;
            if (val_of_foc >= 0 && val_of_foc <= 9)
            {
                bool isfloat = false;
                for (auto x : curr)
                {
                    if (x == '.')
                        isfloat = true;
                }
                if (isfloat)
                {
                    float v = atof(curr.c_str());
                    string p = to_string(floatcc);
                    Tokensed.push_back({"3", p});
                    floatcp[p] = v;
                    floatcc++;
                }
                else
                {
                    int v = stoi(curr);
                    string p = to_string(intcc);
```

```
                    Tokensed.push_back({"2", p});
                    intcp[p] = v;
                    intcc++;
                }
            }
            else
            {
                string p = to_string(idc);

                Tokensed.push_back({"1", p});
                idp[p] = curr;
                idc++;
            }
        }
    }
}

void print_all_Symtabs(map<string, int> &intcp, map<string, float> &floatcp,
map<string, string> &idp)
{
    cout << "The integer constant pointer is: " << endl;
    for (auto x : intcp)
    {
        cout << x.first << "->" << x.second << endl;
    }
    cout << endl;

    cout << "The float constant pointer is: " << endl;
    for (auto x : floatcp)
    {
        cout << x.first << "->" << x.second << endl;
    }
    cout << endl;

    cout << "The identifier pointer is: " << endl;
    for (auto x : idp)
    {
        cout << x.first << "->" << x.second << endl;
    }
    cout << endl;
}

void DisplayPT(vector<vector<string>> productions, map<vector<string>, int> PT)
{
```

```
    cout << "table" << endl;
    ;
}

void Print_1D_vector(vector<string> v)
{
    int n = v.size();
    for (int i = 0; i < n; i++)
    {
        cout << v[i] << " ";
    }
    cout << endl;
}
//-------------------------------------------------------------------------------
------------------

void fillTokens(vector<string> &Tokens, vector<vector<string>> Tokensed)
{
    int n = Tokensed.size();
    for (int i = 0; i < n; i++)
    {
        Tokens.push_back(Tokensed[i][0]);
    }
}
bool OPP(vector<vector<string>> Tokensed)
{
    vector<string> Tokens;
    fillTokens(Tokens, Tokensed);
    cout << "The tokens that we have are:" << endl;
    Print_1D_vector(Tokens);

    // Operator precidecnce table
    map<vector<string>, string> OPT;
    OPT[{"4", "4"}] = ">";
    OPT[{"4", "5"}] = ">";
    OPT[{"4", "6"}] = "<";
    OPT[{"4", "7"}] = "<";
    OPT[{"4", "8"}] = "<";
    OPT[{"4", "1"}] = "<";
    OPT[{"4", "10"}] = "<";
    OPT[{"4", "11"}] = ">";
    OPT[{"4", "$"}] = "<";
    OPT[{"4", "2"}] = "<";
```

```
    OPT[{"5", "4"}] = ">";
    OPT[{"5", "5"}] = ">";
    OPT[{"5", "6"}] = "<";
    OPT[{"5", "7"}] = "<";
    OPT[{"5", "8"}] = "<";
    OPT[{"5", "1"}] = "<";
    OPT[{"5", "10"}] = "<";
    OPT[{"5", "11"}] = ">";
    OPT[{"5", "$"}] = ">";

    OPT[{"6", "4"}] = ">";
    OPT[{"6", "5"}] = ">";
    OPT[{"6", "6"}] = ">";
    OPT[{"6", "7"}] = ">";
    OPT[{"6", "8"}] = "<";
    OPT[{"6", "1"}] = ">";
    OPT[{"6", "10"}] = "<";
    OPT[{"6", "11"}] = ">";
    OPT[{"6", "$"}] = ">";

    OPT[{"7", "4"}] = ">";
    OPT[{"7", "5"}] = "<";
    OPT[{"7", "6"}] = ">";
    OPT[{"7", "7"}] = ">";
    OPT[{"7", "8"}] = "<";
    OPT[{"7", "1"}] = ">";
    OPT[{"7", "10"}] = "<";
    OPT[{"7", "11"}] = ">";
    OPT[{"7", "$"}] = ">";

    OPT[{"8", "4"}] = ">";
    OPT[{"8", "5"}] = ">";
    OPT[{"8", "6"}] = ">";
    OPT[{"8", "7"}] = ">";
    OPT[{"8", "8"}] = ">";
    OPT[{"8", "1"}] = ">";
    OPT[{"8", "10"}] = "<";
    OPT[{"8", "11"}] = ">";
    OPT[{"8", "$"}] = ">";

    OPT[{"1", "4"}] = ">";
    OPT[{"1", "5"}] = "<";
    OPT[{"1", "6"}] = ">";
    OPT[{"1", "7"}] = ">";
```

```
OPT[{"1", "8"}] = ">";
OPT[{"1", "1"}] = ">";
OPT[{"1", "10"}] = "<";
OPT[{"1", "11"}] = ">";
OPT[{"1", "$"}] = ">";
OPT[{"1", "9"}] = "<";

OPT[{"10", "4"}] = "<";
OPT[{"10", "5"}] = "<";
OPT[{"10", "6"}] = "<";
OPT[{"10", "7"}] = "<";
OPT[{"10", "8"}] = "<";
OPT[{"10", "1"}] = "<";
OPT[{"10", "10"}] = "<";
OPT[{"10", "11"}] = "=";
OPT[{"10", "$"}] = "";

OPT[{"11", "4"}] = ">";
OPT[{"11", "5"}] = ">";
OPT[{"11", "6"}] = ">";
OPT[{"11", "7"}] = ">";
OPT[{"11", "8"}] = ">";
OPT[{"11", "11"}] = ">";
OPT[{"11", "$"}] = ">";

OPT[{"$", "4"}] = "<";
OPT[{"$", "5"}] = "<";
OPT[{"$", "6"}] = "<";
OPT[{"$", "7"}] = "<";
OPT[{"$", "8"}] = "<";
OPT[{"$", "1"}] = "<";
OPT[{"$", "$"}] = "<";

OPT[{"3", "4"}] = ">";
OPT[{"3", "5"}] = "<";
OPT[{"3", "6"}] = ">";
OPT[{"3", "7"}] = ">";
OPT[{"3", "8"}] = ">";
OPT[{"3", "3"}] = ">";
OPT[{"3", "10"}] = "<";
OPT[{"3", "11"}] = ">";
OPT[{"3", "$"}] = ">";

OPT[{"2", "4"}] = ">";
```

```
    OPT[{"2", "5"}] = "<";
    OPT[{"2", "6"}] = ">";
    OPT[{"2", "7"}] = ">";
    OPT[{"2", "8"}] = ">";
    OPT[{"2", "2"}] = ">";
    OPT[{"2", "10"}] = "<";
    OPT[{"2", "11"}] = ">";
    OPT[{"2", "$"}] = ">";

    OPT[{"9", "2"}] = "<";
    OPT[{"9", "3"}] = "<";
    OPT[{"9", "1"}] = "<";
    OPT[{"9", "10"}] = "<";
    OPT[{"9", "11"}] = ">";
    OPT[{"9", "$"}] = ">";

    // Defining the grammar
    vector<vector<vector<string>>> grammar;
    grammar.push_back({{"E"}, {"E", "4", "E"}});
    grammar.push_back({{"E"}, {"E", "5", "E"}});
    grammar.push_back({{"E"}, {"E", "6", "E"}});
    grammar.push_back({{"E"}, {"E", "7", "E"}});
    grammar.push_back({{"E"}, {"E", "8", "E"}});
    grammar.push_back({{"E"}, {"1"}});
    grammar.push_back({{"E"}, {"10", "E", "11"}});
    grammar.push_back({{"E"}, {"2"}});
    grammar.push_back({{"E"}, {"3"}});
    grammar.push_back({{"S"}, {"1", "9", "E"}});

    // stack with $ as the bottom element
    stack<string> st;
    st.push("$");

    int n = Tokens.size();

    for (int i = 0; i < n; i++)
    {
        string incoming = Tokens[i];
        // current content of stack
        stack<string> temp = st;
        stack<string> temp2;
        while (!temp.empty())
        {
            temp2.push(temp.top());
```

```
        temp.pop();
    }

    // displaying the current stack

    cout << "The current stack is:" << endl;
    while (!temp2.empty())
    {
        cout << temp2.top() << " ";
        temp2.pop();
    }
    cout << endl;

    // reamining input
    cout << "The remaining input is:" << endl;
    for (int j = i; j < n; j++)
    {
        cout << Tokens[j] << " ";
    }
    cout << endl;

    cout << "----------------------------" << endl;

    while (1)
    {
        string top = st.top();
        // the baic condtion of success
        if(top=="10"){
            return false;
        }

        // condtion when stack top is E and incoming is $
        if (top == "E" && incoming == "$")
        {
            st.pop();
            string newtop = st.top();
            st.push("E");
            top = newtop;

            cout << "The stack top is E and incoming is $" << endl;
            temp = st;
            while (!temp.empty())
            {
                temp2.push(temp.top());
```

```
            temp.pop();
        }

        cout << "The current stack is:" << endl;
        while (!temp2.empty())
        {
            cout << temp2.top() << " ";
            temp2.pop();
        }
        cout << endl;
    }

    if (top == "S" && incoming == "$")
    {
        return true;
    }

    // if the top is less than incoming
    if (OPT[{top, incoming}] == "<")
    {
        st.push(incoming);
        break;
    }

    // if the top is greater than incoming
    if (OPT[{top, incoming}] == ">")
    {
        vector<string> temp;
        while (1)
        {
            string top = st.top();
            st.pop();
            temp.push_back(top);
            if (OPT[{st.top(), top}] == "<")
            {
                break;
            }
        }

        temp.reserve(temp.size());

        // checking if the temp is a valid production
        int m = grammar.size();
        bool found = false;
```

```
            for (int j = 0; j < m; j++)
            {
                if (grammar[j][1] == temp)
                {
                    found = true;
                    st.push(grammar[j][0][0]);
                    cout<<st.top()<<endl;
                    break;
                }
            }

            cout << "The production to find:" << endl;
            for (int i = 0; i < temp.size(); i++)
            {
                cout << temp[i] << " ";
            }
            cout << endl;

            if (!found)
            {
                cout << "The production is not valid" << endl;
                return false;
            }
            if(incoming!="$")
            st.push(incoming);

            if(incoming=="$")
            i--;
            break;
        }

        // if not in precedence table
        if (OPT.find({top, incoming}) == OPT.end())
        {
            return false;
        }
    }
}

if(st.top()=="$")
st.pop();


while(1){
```

```
string special="";
string incoming = "$";
string top = st.top();
// current content of stack
stack<string> temp = st;
stack<string> temp2;
while (!temp.empty())
{
    temp2.push(temp.top());
    temp.pop();
}

// displaying the current stack

cout << "The current stack is:" << endl;
while (!temp2.empty())
{
    cout << temp2.top() << " ";
    temp2.pop();
}
cout << endl;

 if (top == "E" && incoming == "$")
    {
        st.pop();
        string newtop = st.top();
        st.push("E");
        top = newtop;

        cout << "The stack top is E and incoming is $" << endl;
        temp = st;
        while (!temp.empty())
        {
            temp2.push(temp.top());
            temp.pop();
        }

        cout << "The current stack is:" << endl;
        while (!temp2.empty())
        {
            cout << temp2.top() << " ";

            special=special+temp2.top();
```

```
            temp2.pop();
        }

        cout << endl;
        //cout<<"special:"<<special<<endl;
    }

    if(special=="$19E"){
        st.pop();
        st.pop();
        st.pop();

        st.push("S");

        continue;
    }

    if (top == "S" && incoming == "$")
    {
        return true;
    }

    // if the top is less than incoming
    if (OPT[{top, incoming}] == ">")
    {
        // st.push(incoming);
        break;
    }

    // if the top is greater than incoming
    if (OPT[{top, incoming}] == "<")
    {
        vector<string> temp;
        while (1)
        {
            string top = st.top();
            st.pop();
            temp.push_back(top);
            if (st.top()=="9"||OPT[{st.top(), top}] == "<")
            {
                break;
            }
        }
```

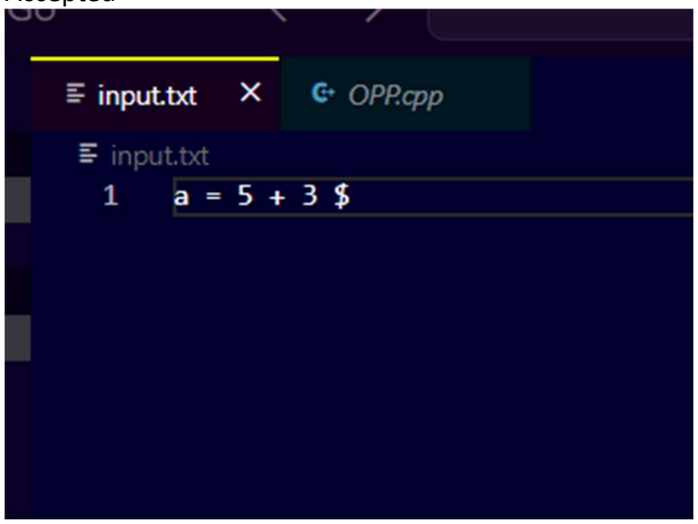
```
                temp.reserve(temp.size());

                // checking if the temp is a valid production
                int m = grammar.size();
                bool found = false;
                for (int j = 0; j < m; j++)
                {
                    if (grammar[j][1] == temp)
                    {
                        found = true;
                        st.push(grammar[j][0][0]);
                        cout<<st.top()<<endl;
                        break;
                    }
                }

                cout << "The production to find:" << endl;
                for (int i = 0; i < temp.size(); i++)
                {
                    cout << temp[i] << " ";
                }
                cout << endl;

                if (!found)
                {
                    cout << "The production is not valid" << endl;
                    return false;
                }




            }

            // if not in precedence table
            if (OPT.find({top, incoming}) == OPT.end())
            {
                return false;
            }
    }

    cout<<"hi"<<endl;
    return false;
}
```

```
//--------------------------------------------------------------------------
--------------------
int main()
{
    // Database starts
    map<string, string> keywords;
    keywords["int"] = "INT";
    keywords["float"] = "FLOAT";
    keywords["+"] = "4";
    keywords["-"] = "5";
    keywords["*"] = "6";
    keywords["/"] = "7";
    keywords["="] = "9";
    keywords["^"] = "8";
    keywords["("] = "10";
    keywords[")"] = "11";
    keywords["$"] = "$"; // End of file
    // 3 for float
    // 2 for int

    // 1 for identifiers

    // pointer to intc
    map<string, int> intcp;
    // pointer ot intf
    map<string, float> floatcp;
    // pointer to identifier
    map<string, string> idp;

    // Database ends
    string inputFile;
    vector<string> input;
    vector<vector<string>> Tokensed;

    cout << "Enter the name of the file: ";
    cin >> inputFile;
    readfile(inputFile, input);

    cout << "The input file is: " << endl;
    print_vector(input);

    Tokenization(input, Tokensed, keywords, intcp, floatcp, idp);
    cout << "The tokens are: " << endl;
```

```
    print_vector_2D(Tokensed);

    print_all_Symtabs(intcp, floatcp, idp);

    // Logic of Parser

    bool Parsed = OPP(Tokensed);

    if (Parsed)
    {
        cout << "The string follows the grammer" << endl;
    }
    else
    {
        cout << "The grammer is not followed" << endl;
    }

    return 0;
}
```

End Result:

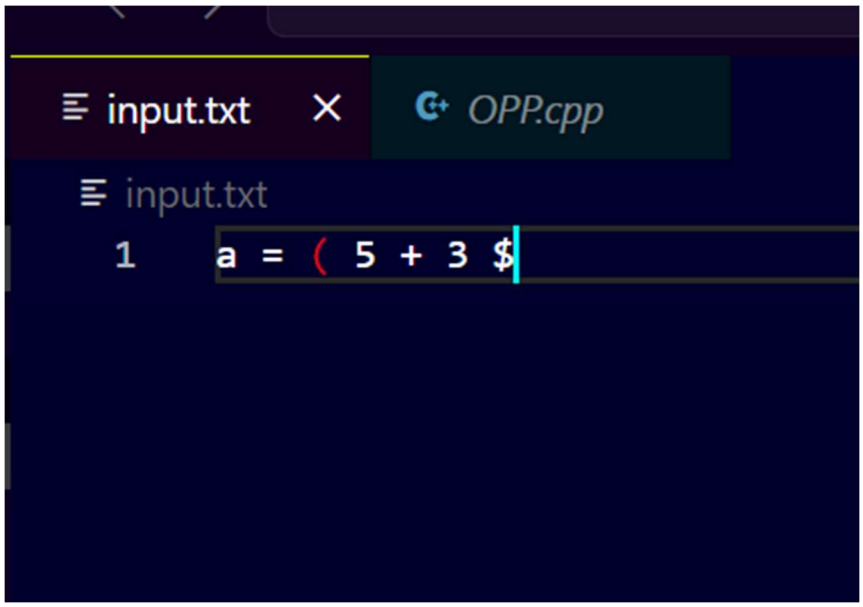
Accepted=>



```
PS E:\GIt\SEM-6\SPCC\Compiler> cd "e:\GIt\SEM-6\SPCC\Compiler\" ; i
Enter the name of the file: input.txt
The input file is:
a = 5 + 3 $
The tokens are:
1 *->0
9 *->NA
2 *->0
4 *->NA
2 *->1
$ *->NA

The integer constant pointer is:
0->5
1->3

The float constant pointer is:

The identifier pointer is:
0->a

The tokens that we have are:
1 9 2 4 2 $
The current stack is:
$
The remaining input is:
1 9 2 4 2 $
----------------------------
The current stack is:
$ 1
The remaining input is:
9 2 4 2 $
----------------------------
The current stack is:
$ 1 9
The remaining input is:
2 4 2 $
----------------------------
The current stack is:
$ 1 9 2
The remaining input is:
4 2 $
----------------------------
```

```
$ 1 9 E 4 2
The remaining input is:
$
----------------------------
E
The production to find:
2
The current stack is:
$ 1 9 E 4 E
The remaining input is:
$
----------------------------
The stack top is E and incoming is $
The current stack is:
$ 1 9 E 4 E
The current stack is:
$ 1 9 E 4 E
The stack top is E and incoming is $
The current stack is:
$ 1 9 E 4 E
E
The production to find:
E 4 E
The current stack is:
$ 1 9 E
The stack top is E and incoming is $
The current stack is:
$ 1 9 E
The current stack is:
$ S
The string follows the grammer
PS E:\GIt\SEM-6\SPCC\Compiler>
```

Not Accepted=>


input.txt
OPP.cpp
input.txt
1
a = (5 + 3 $

```
The input file is:
a = ( 5 + 3 $
The tokens are:
1 *->0
9 *->NA
10 *->NA
2 *->0
4 *->NA
2 *->1
$ *->NA

The integer constant pointer is:
0->5
1->3

The float constant pointer is:

The identifier pointer is:
0->a

The tokens that we have are:
1 9 10 2 4 2 $
The current stack is:
$
The remaining input is:
1 9 10 2 4 2 $
----------------------------
The current stack is:
$ 1
The remaining input is:
9 10 2 4 2 $
----------------------------
The current stack is:
$ 1 9
The remaining input is:
10 2 4 2 $
----------------------------
The current stack is:
$ 1 9 10
The remaining input is:
2 4 2 $
----------------------------
The grammer is not followed
PS E:\GIt\SEM-6\SPCC\Compiler>
```