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

Student Name	Deep Salunkhe
Roll Number	21102A0014
TE Division	A

Title:

LL1 Parser

Approach:

1. **File Reading Function (readfile)**:
    - This function reads input from a file specified by the user.
    - It tokenizes the input based on spaces and newlines, storing each token in a vector.
2. **Tokenization Function (Tokenization)**:
    - This function takes the vector of tokens generated by **readfile** and further processes them.
    - It identifies keywords, integers, floats, and identifiers, assigning them appropriate tokens for further processing.
    - Tokens are stored in a 2D vector, where each token is represented by a pair of strings - the type of token and its value.
3. **LL1 Parsing Function (LL1Parser)**:
    - This function performs LL(1) parsing on the tokenized input.
    - It uses a predefined parsing table (**PT**) to guide the parsing process.

- The parsing table contains rules for deriving non-terminals from terminals.
- It employs a stack-based approach, where the stack contains symbols to be processed, and the input is consumed from left to right.
- At each step, the function pops the top symbol from the stack and matches it with the current input symbol.
- Based on the match and the parsing table, it either continues parsing or reports an error.
- The parsing process continues until the input is exhausted and the stack is empty.

4. **Main Function**:
    - In the main function, the program interacts with the user to get the input file name.
    - It reads the input file, tokenizes the input, and performs LL(1) parsing.
    - Finally, it prints whether the input string follows the grammar rules or not.

Comparison:

1. **LL(0) Parser**:
    - The LL(0) parser functions consist of recursive descent functions for each non-terminal symbol in the grammar.
    - Each function attempts to match the input with the production rules associated with its respective non-terminal symbol without considering lookahead tokens.
    - Backtracking is employed in LL(0) parsing functions to handle multiple possible choices for production rules.
    - For example, the **F** function in the LL(0) parser examines the current token without considering the next token (lookahead), making decisions solely based on the current token and backtracking if necessary.
2. **LL(1) Parser**:
    - The LL(1) parser function (**LL1Parser**) utilizes a parsing table (**PT**) to guide the parsing process.
    - The parsing table is constructed based on the grammar's production rules and provides information on which production rule to apply for each combination of non-terminal and terminal symbols.
    - LL(1) parsing employs a stack-based approach and considers both the current symbol on the stack and the next input token (lookahead) to determine which production rule to apply.

- For example, in the **LL1Parser** function, the parser examines both the top symbol on the stack and the current input token to decide which production rule to apply, facilitating predictive parsing without backtracking.

3. **Comparison**:

   - **Lookahead**: LL(0) parsers do not use lookahead, while LL(1) parsers use one symbol of lookahead, allowing for more predictive parsing decisions.

   - **Parsing Strategy**: LL(0) parsers employ recursive descent without the need for a parsing table, while LL(1) parsers use parsing tables for predictive parsing.

   - **Complexity**: LL(1) parsers can handle a broader class of grammars due to lookahead but may require more complex parsing table construction.

   - **Error Handling**: LL(1) parsers provide more detailed error messages due to lookahead capabilities, while LL(0) parsers may have more limited error handling.

4. Time and Space complexity

1. **LL(0) Parser**:

   - **Time Complexity**:

     - The time complexity of the LL(0) parser largely depends on the size of the input and the structure of the grammar.

     - In the worst-case scenario, where backtracking is required at each step, the time complexity can be exponential.

     - However, for LL(0) grammars without ambiguity, the time complexity is generally linear or close to linear with respect to the size of the input.

   - **Space Complexity**:

     - The space complexity of the LL(0) parser depends on the depth of the recursive function calls and the size of the stack.

     - In the worst-case scenario, where backtracking leads to deep recursion, the space complexity can be exponential.

     - However, for LL(0) grammars with limited recursion and backtracking, the space complexity is generally linear with respect to the size of the input.

2. **LL(1) Parser**:

   - **Time Complexity**:

     - The time complexity of the LL(1) parser is generally linear with respect to the size of the input.

- Parsing decisions are made based on the information stored in the parsing table, leading to efficient parsing without backtracking.

- **Space Complexity**:
  - The space complexity of the LL(1) parser depends on the size of the parsing table and the depth of the stack during parsing.
  - The size of the parsing table is proportional to the number of non-terminals and terminals in the grammar and the number of production rules.
  - The depth of the stack during parsing depends on the structure of the input and the grammar.
  - Overall, the space complexity of the LL(1) parser is generally linear with respect to the size of the input and the size of the parsing table.

Implementation:

```cpp
#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;
    ;
```

```
}

bool LL1Parser(vector<vector<string>> Tokensed)
{

    // Parser table

    vector<vector<string>> productions = {
        {"1", "9", "E"},    // 0
        {"T", "E_"},        // 1
        {"4", "T", "E_"},   // 2
        {"5", "T", "E_"},   // 3
        {"P", "T_"},        // 4
        {"6", "P", "T_"},   // 5
        {"7", "P", "T_"},   // 6
        {"F", "P_"},        // 7
        {"8", "P"},         // 8
        {"1"},              // 9
        {"10", "E", "11"}, // 10
        {"2"},              // 11
        {"3"},              // 12
        {"0"},              // 13

    };

    // Parser table
    map<vector<string>, int> PT;
    PT[{"S", "1"}] = 0;
    PT[{"E", "1"}] = 1;
    PT[{"T", "1"}] = 4;
    PT[{"P", "1"}] = 7;
    PT[{"F", "1"}] = 9;
    PT[{"E_", "4"}] = 2;
    PT[{"T_", "4"}] = 13;
    PT[{"P_", "4"}] = 13;
    PT[{"T_", "5"}] = 13;
    PT[{"E_", "5"}] = 3;
    PT[{"P_", "5"}] = 0;
    PT[{"T_", "6"}] = 5;
    PT[{"P_", "6"}] = 13;
    PT[{"T_", "7"}] = 6;
    PT[{"P_", "7"}] = 13;
    PT[{"P_", "8"}] = 8;
    PT[{"E", "10"}] = 1;
```

```
    PT[{"T", "10"}] = 4;
    PT[{"P", "10"}] = 7;
    PT[{"F", "10"}] = 10;
    PT[{"E_", "11"}] = 13;
    PT[{"T_", "11"}] = 13;
    PT[{"P_", "11"}] = 13;
    PT[{"E", "2"}] = 1;
    PT[{"E", "3"}] = 1;
    PT[{"T", "2"}] = 4;
    PT[{"T", "3"}] = 4;
    PT[{"P", "2"}] = 7;
    PT[{"P", "3"}] = 7;
    PT[{"F", "2"}] = 11;
    PT[{"F", "3"}] = 12;
    PT[{"E_", "$"}] = 13;
    PT[{"T_", "$"}] = 13;
    PT[{"P_", "$"}] = 13;

    // DisplayPT(productions,PT);

    string Inputstring = "";
    for (auto x : Tokensed)
    {

        Inputstring = Inputstring + x[0];
    }

    cout << "The input string is:" << endl;
    cout << Inputstring << endl;

    stack<string> s;
    s.push("S");
    int pin = 0;
    bool success = false;

    while (s.empty() == false && pin < Inputstring.size())
    {

        cout << "Stack top: " << s.top() << endl;
        cout << "Input string: " << Inputstring[pin] << endl;

        // condition for epsilon

        if (s.empty() == false && s.top() == "0")
```

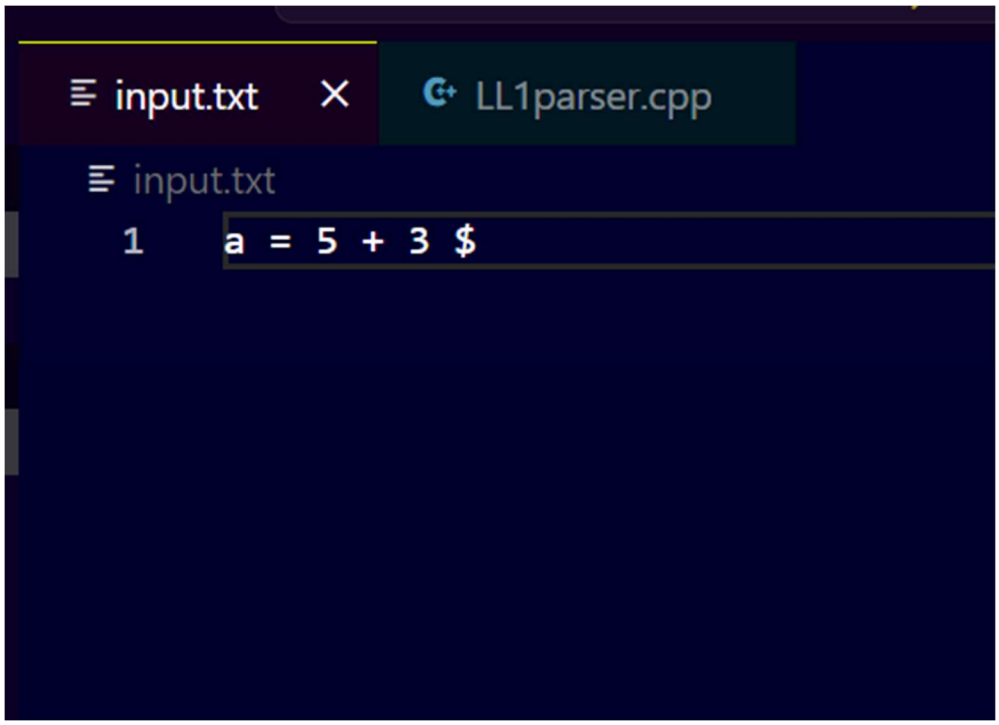
```
        {
            s.pop();
            if(s.empty() == true && Inputstring[pin] == '$')
            {
                success = true;
                break;
            }
            continue;
        }

        if (Inputstring[pin] == '$' && s.empty() == true)
        {
            success = true;
            break;
        }
        string some = string(1, Inputstring[pin]);
        if (s.top() == some)
        {
            s.pop();
            pin++;

            continue;
        }

        string stop = s.top();
        string ic = string(1, Inputstring[pin]);

        if (PT.find({stop, ic}) != PT.end())
        {

            s.pop();
            vector<string> temp = productions[PT[{stop, ic}]];
            int ts = temp.size();
            for (int i = ts - 1; i >= 0; i--)
            {
                s.push(temp[i]);
            }
        }
        else
        {
            success = false;
            break;
        }
    }
```

```
    return success;
}

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 iscorrect = LL1Parser(Tokensed);
    if (iscorrect)
    {
        cout << "The format grammer is follow" << endl;
    }
    else
    {
        cout << "The grammer is not followed" << endl;
    }

    return 0;
}
```

End Result:

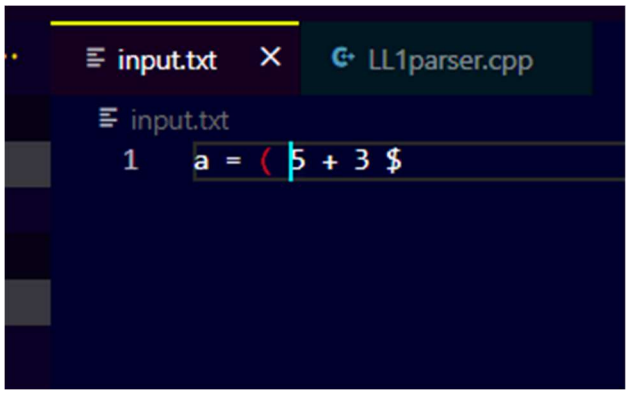
Accepted=>


input.txt
LL1parser.cpp
input.txt
1
a = 5 + 3 $

```
PS E:\GIt\SEM-6\SPCC\Compiler> cd "e:\GIt\SEM-6\SPCC\Compiler\" ; if ($?) { g++ LL1parse
\LL1parser }
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 input string is:
19242$
Stack top: S
Input string: 1
Stack top: 1
Input string: 1
Stack top: 9
Input string: 9
```

```
Input string: 1
Stack top: 9
Input string: 9
Stack top: E
Input string: 2
Stack top: T
Input string: 2
Stack top: P
Input string: 2
Stack top: F
Input string: 2
Stack top: 2
Input string: 2
Stack top: P_
Input string: 4
Stack top: 0
Input string: 4
Stack top: T_
Input string: 4
Stack top: 0
Input string: 4
Stack top: E_
Input string: 4
Stack top: 4
Input string: 4
Stack top: T
Input string: 2
Stack top: P
Input string: 2
Stack top: F
Input string: 2
Stack top: 2
Input string: 2
Stack top: P_
Input string: $
Stack top: 0
Input string: $
Stack top: T_
Input string: $
Stack top: 0
Input string: $
Stack top: E_
Input string: $
Stack top: 0
Input string: $
The format grammer is follow
```

Not Accepted=>



```
Enter the name of the file: input.txt
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 input string is:
1910242$
Stack top: S
Input string: 1
Stack top: 1
Input string: 1
Stack top: 9
Input string: 9
Stack top: E
Input string: 1
Stack top: T
Input string: 1
Stack top: P
Input string: 1
Stack top: F
Input string: 1
Stack top: 1
Input string: 1
Stack top: P_
Input string: 0
The grammer is not followed
```