```
// STUDENT NUMBER : B231202072
// STUDENT NAME : BEYZA KOMISH
// LR PARSER HOMEWORK-1


//!!!!!!!!!!!!!!!!!!!!
```

```
//HONOR CODE :
//ALL THE LOGIC HAS BEEN LEARNED AND IMPLEMENTED BY BEYZA KOMISH FOR CLARIFICATION GOT HELP
FROM RECOURCES WHEN LOGGING , READING THE FILES
// AND ADJUSTED CODE READABILITY
```

```
//!!!!!!!!!!!!!!!!!!!!
```

```
// please be located in the output folder and run in the terminal these commands below:
//   g++ -o lrParser.exe lrParser.cpp       --for compilation
//  ./lrParser.exe                          --for runnung the executable
```

Introduction

This report aims to explain the purpose and functionality of the parser logic implemented in the program. The parser's job is to process an input sequence, determine if it follows a specific grammar, and generate a parse tree representing the structure of the input.

The project has code comments to help understand the implementaion better.

The Logic Behind the Parser

The parser operates within a while loop that continuously processes input tokens. The loop stops when one of the following conditions is met:
- Accept: The input string is successfully parsed, meaning the grammar rules are correctly followed.
- Error: An unknown token or syntax error is encountered.
- Shift or Reduce: Based on the action specified in the action table, the parser either shifts a token onto the stack or reduces them based on a grammar rule.

Understood the Shift and Reduce actions

The key operations that the parser performs are shift and reduce.

-Shift: When the action table directs the parser to shift, the current input token is pushed onto the stack along with the new state number. This means the parser is moving to the next token in the sequence and adding it to the stack for future processing.

```
Stack.push(currentInput);
Stack.push(action.substr(1)); // Push new state
```

- Reduce: When the action table directs the parser to reduce, the parser pops items off the stack (the right-hand side of a grammar rule) and pushes the left-hand side of the rule onto the stack. This simulates applying a production rule to simplify the parse process.

```
Stack.pop();  // Pop tokens from stack according to the rule
Stack.push(rule.lhs);  // Push the left-hand side of the rule
```

Shift Logic

When the action table specifies a shift check with the if case : `Stack.push(currentInput);`

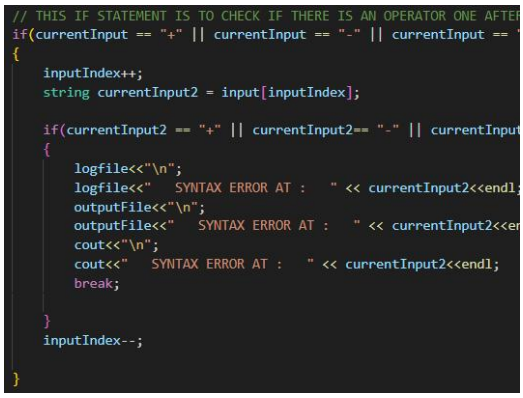
```
else if ( action[0] == 's' )//if the cell starts with s which means the action 'SHIFT'
          {

              Stack.push(action.substr(1));

               Stack.push(currentInput);
               Stack.push(action.substr(1));
```

the parser logs the current state of the stack, the current input token, and the action. It then shifts the token onto the stack and moves to the next token.

The shift process also checks for syntax errors—if two operators appear consecutively it triggers an error:



Reduce Logic

When the action table specifies a reduce handled with the if case:

the parser logs the current state of the stack, the rule being applied, and the grammar. It then pops elements from the stack corresponding to the right hand side of the grammar rule, and pushes the left hand side onto the stack.

```
int length = rule.rhs.size()*2;
               for (int i = 0; i < length; i++)
               {
                   Stack.pop();
               }

               int stackTop = stoi(Stack.top());

               int newState =  gotoTable[stackTop][rule.lhs];

               Stack.push(rule.lhs);
               Stack.push(to_string(newState));
```

The reduction creates a node in the parse tree, which is then pushed onto a separate parse stack.

Logging and Debugging

To help with debugging and understanding the parsing process, detailed logs are created at each step. These logs provide insights into the actions taken, the state of the stack, the grammar rule applied, and the resulting changes in the parse tree.

For example, when shifting, the program logs:

- The current stack content
- The current state and input token
- The shift action taken and the state it moved to

Similarly, during a reduce, the program logs:
- The state before the reduce
- The grammar rule used
- The state after the reduce and the new item pushed to the stack

```
                logfile << "Shift and goto state " << action.substr(1) << ": push the token
\"" << currentInput << "\" and state " << action.substr(1) << " onto the stack" << endl;
                logfile << "Parse tree: token \"" << currentInput << "\" becomes a node
(leaf)" << endl;
                logfile << "Move to next token" << endl;
                logfile<<"\n";
```

Handling Errors

Errors are handled in two major cases:

Unknown Tokens: If the current input token is unknown (i.e., not in the action table), the program will log an error and break out of the parsing loop.

```
if( currentInput == "-")
{
    logfile<<"\n";
    logfile<<"   UNKOWN TOKEN  :   " << currentInput<<endl;
    outputFile<<"\n";
    outputFile<<"   UNKOWN TOKEN  :   " << currentInput<<endl;
    cout<<"\n";
    cout<<"   UNKOWN TOKEN  :   " << currentInput<<endl;

    break;
}
```

2.Syntax Errors: If the parser detects consecutive operators or mismatched input based on the grammar, a syntax error is logged, and the parsing is halted.

```
                // THIS IF STATEMENT IS TO CHECK IF THERE IS AN OPERATOR ONE AFTER THE OTHER WHICH
IS A SYNTAX ERROR,PRINT ANY SHIFT COMMANDS
                if(currentInput == "+" || currentInput == "-" || currentInput == "*" || currentInput
== "/")
                {
                    inputIndex++;
                    string currentInput2 = input[inputIndex];

                    if(currentInput2 == "+" || currentInput2== "-" || currentInput2 == "*" ||
currentInput2 == "/")
                    {
                        logfile<<"\n";
                        logfile<<"   SYNTAX ERROR AT :   " << currentInput2<<endl;
                        outputFile<<"\n";
                        outputFile<<"   SYNTAX ERROR AT :   " << currentInput2<<endl;
                        cout<<"\n";
                        cout<<"   SYNTAX ERROR AT :   " << currentInput2<<endl;
                        break;

                    }
                    inputIndex--;

                }
```

Void Functions for Reading and Filling Data Structures

In order to effectively manage the parsing process, we rely on void functionsto read the input, populate the action table, goto table, and input sequence. These functions ensure that the necessary data structures are filled with the appropriate values, setting up the parser with the required resources for processing.

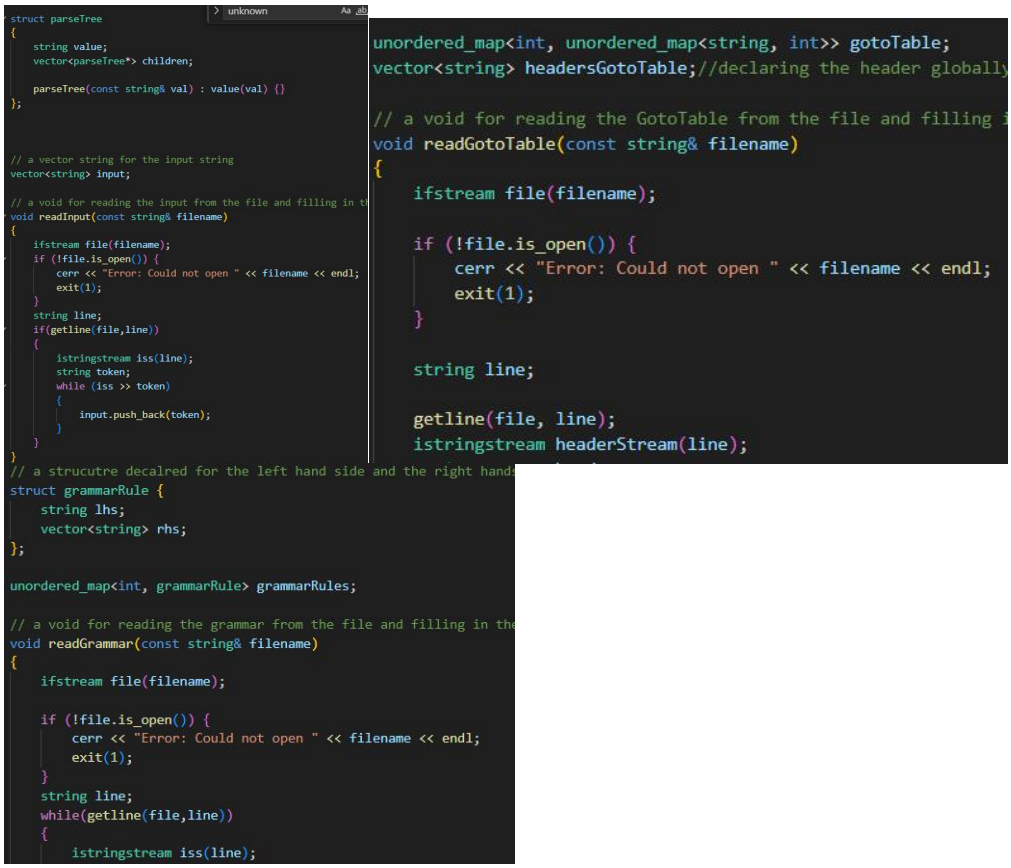
Why These Data Structures?

- action Table: The action table is essential for guiding the parser's decision-making process. It helps the parser determine what action to take at each point based on the current state and input token. I chose to implement the action table as a unordered_map because it allows efficient lookups, ensuring that the parser can quickly retrieve the appropriate action for any given state and token.

- Goto Table The goto table tracks the state transitions for non-terminals after reductions. Like the action table, it's implemented as a unordered map, where the key is the non-terminal symbol, and the value is the state it transitions to. This structure is chosen to optimize access times for state transitions.

- Input vector: The input sequence is stored in a vector for easy access by the parser. The vector allows for straightforward indexing into the input and easy manipulation of tokens during the parsing process.

The choice of these data structures—hash maps which are unordered maps for the action and goto tables, and a vector for the input sequence—was made to optimize both time and space efficiency. The unordered_map ensures fast lookups and inserts, which is crucial for handling the large number of states and tokens in the action and goto tables. The vector provides constant-time random access to the input sequence, allowing the parser to process tokens quickly.



```
struct parseTree
{
    string value;
    vector<parseTree*> children;

    parseTree(const string& val) : value(val) {}
};



// a vector string for the input string
vector<string> input;

// a void for reading the input from the file and filling in t
void readInput(const string& filename)
{
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Error: Could not open " << filename << endl;
        exit(1);
    }
    string line;
    if(getline(file,line))
    {
        istringstream iss(line);
        string token;
        while (iss >> token)
        {
            input.push_back(token);
        }
    }
}
// a strucutre decalred for the left hand side and the right hands
struct grammarRule {
    string lhs;
    vector<string> rhs;
};

unordered_map<int, grammarRule> grammarRules;

// a void for reading the grammar from the file and filling in the
void readGrammar(const string& filename)
{
    ifstream file(filename);

    if (!file.is_open()) {
        cerr << "Error: Could not open " << filename << endl;
        exit(1);
    }
    string line;
    while(getline(file,line))
    {
        istringstream iss(line);
```

```
unordered_map<int, unordered_map<string, int>> gotoTable;
vector<string> headersGotoTable;//declaring the header globally

// a void for reading the GotoTable from the file and filling i
void readGotoTable(const string& filename)
{
    ifstream file(filename);

    if (!file.is_open()) {
        cerr << "Error: Could not open " << filename << endl;
        exit(1);
    }

    string line;

    getline(file, line);
    istringstream headerStream(line);
```

```
readInput("input.txt");
readGrammar("Grammar.txt");
readGotoTable("GotoTable.txt");
readActionTable("ActionTable.txt");
```

Accepting Input and Generating the Parse Tree

When the input is accepted (i.e., the parsing process completes successfully), the program:
- Logs that the input was accepted
- Generates the parse tree by printing the tree structure

This parse tree is an important part of the program, showing how the input fits into the grammar defined by the language.

logfile << "Parse tree generated" << endl;
printParseTree(root, "", logfile); // Print parse tree to log file

```
void printParseTree(parseTree* node, string pathSoFar, ofstream& logfile) {
    if (!node) return;

    pathSoFar += "/" + node->value;
    logfile << pathSoFar << endl;

    for (auto child : node->children) {
        printParseTree(child, pathSoFar, logfile);
    }
```

Conclusion

The parser is designed to simulate the process of analyzing a sequence of tokens using a shift-reduce approach. It effectively handles shifting and reducing according to grammar rules, and ensures error detectio and logging for both unknown tokens and syntax errors. The program generates a parse tree once the input is successfully parsed, providing a structured representation of the input.

The parser's use of a stack, action table, and grammar rules enables it to process complex sequences and ensure they conform to the defined grammar. The detailed logging ensures that each step of the process is transparent and traceable, aiding in debugging and understanding the parsing workflow.

The void functions for reading and filling in the action table, goto table, and input sequence are crucial in preparing the parser for efficient operation. The choice of unordered_map and vectorfor storing these data structures ensures that the parser operates in an optimal manner with fast lookups and minimal overhead.

```
ACCEPTED

Parse tree generated
/E
/E/T
/E/T/T
/E/T/T/F
/E/T/T/F/id
/E/T/*
/E/T/F
/E/T/F/(
/E/T/F/E
/E/T/F/E/E
/E/T/F/E/E/T
/E/T/F/E/E/T/F
/E/T/F/E/E/T/F/id
/E/T/F/E/+
/E/T/F/E/T
/E/T/F/E/T/F
/E/T/F/E/T/F/id
/E/T/F/)
```

input.txt | lrParser.cpp | output.txt

output > input.txt

```
1   id * ( id + id ) $
```

input.txt | lrParser.cpp | output.txt | Log.txt

output > output.txt

Step	Stack	Input Left	Action
0	0 id 5	id * (id + id) $	Shift 5
1	0 F 3	* (id + id) $	Reduce 6 F -> id (GOTO[0, F])
2	0 T 2	* (id + id) $	Reduce 4 T -> F (GOTO[0, T])
3	0 T 2 * 7	(id + id) $	Shift 7
4	0 T 2 * 7 (4	id + id) $	Shift 4
5	0 T 2 * 7 (4 id 5	+ id) $	Shift 5
6	0 T 2 * 7 (4 F 3	+ id) $	Reduce 6 F -> id (GOTO[4, F])
7	0 T 2 * 7 (4 T 2	+ id) $	Reduce 4 T -> F (GOTO[4, T])
8	0 T 2 * 7 (4 E 8	+ id) $	Reduce 2 E -> T (GOTO[4, E])
9	0 T 2 * 7 (4 E 8 + 6	id) $	Shift 6
10	0 T 2 * 7 (4 E 8 + 6 id 5	) $	Shift 5
11	0 T 2 * 7 (4 E 8 + 6 F 3	) $	Reduce 6 F -> id (GOTO[6, F])
12	0 T 2 * 7 (4 E 8 + 6 T 9	) $	Reduce 4 T -> F (GOTO[6, T])
13	0 T 2 * 7 (4 E 8	) $	Reduce 1 E -> E + T (GOTO[4, E])
14	0 T 2 * 7 (4 E 8) 11	) $	Shift 11
15	0 T 2 * 7 F 10	$	Reduce 5 F -> (E) (GOTO[7, F])
16	0 T 2	$	Reduce 3 T -> T * F (GOTO[0, T])
17	0 E 1	$	Reduce 2 E -> T (GOTO[0, E])
18	0 E 1	$	ACCEPT

```
Action table loaded:
------------------------------------------------------------
State   id  +   *   (   )   $
------------------------------------------------------------
0   s5  -   -   s4  -   -
1   -   s6  -   -   -   accept
2   -   r2  s7  -   r2  r2
3   -   r4  r4  -   r4  r4
4   s5  -   -   s4  -   -
5   -   r6  r6  -   r6  r6
6   s5  -   -   s4  -   -
7   s5  -   -   s4  -   -
8   -   s6  -   -   s11 -
9   -   r1  s7  -   r1  r1
10  -   r3  r3  -   r3  r3
11  -   r5  r5  -   r5  r5
------------------------------------------------------------
Goto table loaded:
------------------------------------------------------------
State     E         T         F
0         1         2         3
1         -         -         -
2         -         -         -
3         -         -         -
4         8         2         3
5         -         -         -
6         -         9         3
7         -         -         10
8         -         -         -
9         -         -         -
10        -         -         -
11        -         -         -
------------------------------------------------------------
Grammar loaded:
------------------------------------------------------------
ID   LHS        RHS
------------------------------------------------------------
1    E    ->  E + T
2    E    ->  T
3    T    ->  T * F
4    T    ->  F
5    F    ->  ( E )
6    F    ->  id
------------------------------------------------------------
Parsing for the input  : id * ( id + id ) $

------------------------------------------------------------
Current stack content: 0
Current state (stack-top) is 0, next token is 'id' (action table column 0)
Fetched action from the table: s5

Shift and goto state 5: push the token "id" and state 5 onto the stack
Parse tree: token "id" becomes a node (leaf)
Move to next token


Current stack content: 0 id 5
Current state (stack-top) is 5, next token is '*' (action table column 1)
Fetched action from the table: r6


Reduce using grammar rule 6: F -> id
Stack content before reduce: 0 id 5
Pop tokens and associated states from the stack considering the tokens at RHS of grammar rule (6: F -> id )


State at stack is 0 (current stack content: 0 )
Push LHS (F) of the grammar rule (6: F -> id ) onto stack
```