1. **Problem Statement**

The third assignment is to implement (or add implementation to) one of the following parsers: Bottom-Up, Top-Down RDP, Top-Down using a Stack, or a different approach than the one used for the last project. In addition, students must also implement symbol table handling with type checking or a parse tree implementation. More points will be achieved by implementing the following: scope checking, type checking, testing, error handling, and/or integrating all the assignments to work together.

1. **How to use the program**

a. Using the Windows Command Prompt:

Extract the folder on the Desktop: Compiler

Locate the extracted folder, PRDP, from the terminal using ‘cd Desktop/Compiler’

Compile the program using: g++ -std=c++14 main.cpp -o main -Wl,--stack,268435456

Type, main.exe, to run the program

Note:

Enter only the test file names i.e., ‘v2.txt’

b. Using the Executable File

Extract the folder: Compiler

Double click on the extracted folder to open it

Double click on the executable file: main.exe

The extracted Compiler folder should contain all files listed below to run correctly:

└─────────── **Compiler/**

| └──────── **files/**

| | └──────── **out.txt**

| | └──────── **e1.txt**

| | └──────── **e2.txt**

| | └──────── **v1.txt**

| | └──────── **v2.txt**

| | └──────── **v3.txt**

| | └──────── **v4.txt**

| |

| └──────── **src/**

| | └──────── **Lexer/**

| | └──────── LEXER.h

| | └──────── **Syntax/**

| | └──────── LR.h

| | └──────── PRDP.h

| | └──────── STACK.h

| | └──────── **Tables/**

| | └──────── error.h

| | └──────── symbol.h

| | └──────── tree.h

| └──────── **main.cpp**

| └──────── **main.exe**

My program had already navigated into the test folder directory; therefore, you would only need to type the test file names with the file extension, i.e., ‘1.txt’. My program allows for console navigation. Thus, all test files can be tested by running the executable file once. Once you have successfully opened a text file, you will be prompted with a table structure that asks for user input. Keep in mind only for the menu; your keystroke will be automatically read. Hence, no <Enter> key will be required after input. You can view all tokens and lexemes by hitting the number 1. There is an additional option for the syntax analyzer to turn on productions by hitting the number 0. This allows the user to control their output. If view productions are turned on, the output file will print a list of all productions for each token that is accepted by the parser. Otherwise, the output file will only contain the list of accepted lexemes. You can also access the symbol table by hitting the number 3. You will be presented with another table. The next section will describe the functionalities of the implementations.

**3. Design of the program**

Console Navigation State Diagram:

***LEXER***

Added

1. Inheritance – with Symbol and Error Tables
2. Default Constructor – passes the current file name being read
3. Renamed: insert() -> store() and display() -> printToken()
4. Removed: viewTable() since the Lexer class can access the Symbol Tables protected member variables
5. EOS (%) – after inserting all tokens and lexemes, the Lexer will add an additional token, end of string character, to help processing in later stages

Functionality and Design is the same according to assignment’s 1 and 2 documentations.

***PRDP***

Added:

1. #define type, scope as every token will contain a particular type and scope reference
2. Inheritance – with Symbol, Error, and Tree Tables
3. Default Constructor – passes the current file name being read

|  |  |  |
| --- | --- | --- |
| **Variables** | **Types** | **Definitions** |
| save | Token | the memory address of the LHS identifier in the assignment rule, so the program knows where to pop. |
| jump\_stack | stack<size\_t> | a temporary memory location of a looping or block statement used to jump to the memory location given a boolean value of the conditional statement. |
| CURR\_TYPE | string | saves the type value whenever a declaration statement is called. In addition, when a valid identifier is found in the declaration statement, its type value will be assigned to the CURR\_TYPE. This also expands to the MID production rule as well. |
| SAVE\_TYPE | string | a temporary type value referencing an identifier whenever an assignment, expression and/or conditional statement is called. When a valid identifier is found, it will call the inType function, determining if the type value matches other identifiers in that statement. At the end of those statements, it will reset the type value to an empty string to indicate a new type value should be assigned whenever those statements are called. |
| CURR\_SCOPE | size\_t | holds the current scope reference value starting from zero as the global scope. If the parser enters a block statement, either IF, WHILE, or BEGIN, it will increment the scope by one. After the ending block statement is called, either endif, whileend, or end, it will decrement the scope by one. In addition, whenever an identifier is called, the current scope value will be inserted into the symbol table. Of course, the symbol table will verify for double declarations, undeclared tokens, out of scope, etc., before inserting the current scope value. |
| SAVE\_ADDR | size\_t | a temporary memory address value whenever a looping construct is called. It will be assigned to the op-code ‘JUMP’ and used in the backPatch function. |
| OP | char | a char value holding a relational operator used to find the correct instruction, either LES or GRT, in this grammar. |

|  |  |
| --- | --- |
| **Functions** | **Definitions** |
| void backPatch(const size\_t&) | finds the operand memory address from the top of the stack and replaces it with the current address, aLoc. In detail, it will jump to the previous memory address of the looping construct if the conditional statement returns true. |
| void printAL() | prints all assembly instructions containing the address, op-code and operand. |
| void getTree() | prints a parse tree after concluding the statement is syntactically correct. |
| Node<string>\* getRoot() | returns the root node in the Tree class. |
| void makeTree(T, const vector<T>&) | calls the makeTree function every time printRules() is called. The first argument pass is a string representing a nonterminal. The second argument passes in a vector of child nodes. |

Functionality and Design is the same according to assignment’s 1 and 2 documentations

***Stack***

|  |  |  |
| --- | --- | --- |
| **Variables** | **Types** | **Definitions** |
| output | ofstream | contains all information that will output to a file. |
| ss | stack<string> | holds states and symbols that will be pushed and/or popped according to the rule from the table (rules). |
| on | bool | holds a boolean value that will indicate whether the program should print the production rules or not. |
| t | string | references to the top of the stack. |
| r | size\_t | holds the current indexed rule from the table (rules). |
| table | vector<Token> | a structure that holds all information, Token, of every valid declaration of an identifier. |
| rules | vector<vector<int>> | a two-dimensional dynamic array structure that determines which rule to push onto the stack. |

|  |  |
| --- | --- |
| **Functions** | **Definitions** |
| Node<string>\* getRoot() | returns the root node from the Tree class after returning true from the stack parser. |
| void getTree() | a helper function that prints the parse tree. |
| void debug(stack<string>, symbol) | prints out all iterations to the console. |
| void copy(symbol&) | copies the symbol table generated from the Lexer. |
| bool parser(symbol, bool) | parses the entire string to validate whether the string is syntactically correct or not. It will parse until the stack is empty. |
| bool IS\_TERMINAL(string) | returns a boolean value whether the input string is a terminal in the grammar. |
| int TERMINAL(string) | returns an integer value corresponding to the column of the table (rules). |
| int NONTERMINAL(string) | returns an integer value corresponding to the row of the table (rules). |

Functionality and Design is the same according to assignment’s 1 and 2 documentations.

***LR:***

|  |  |  |
| --- | --- | --- |
| **Variables** | **Types** | **Definitions** |
| output | ofstream | contains all information that will output to a file. |
| rtable | vector<vector<string> | the parsing table containing states, terminals + %, nonterminals where each column contains either shifts and reduce to evaluate whether an input string is syntactically correct or not. |
| ss | stack<string> | holds states and symbols that will be pushed and/or popped according to the rule from the rtable. |
| X | string | holds the current state. |
| TOS | string | references to the top of the stack |

|  |  |
| --- | --- |
| **Functions** | **Definitions** |
| int rules (const int&) | returns the number of elements from a given production rule. |
| string LHS(const string&) | returns the left-hand side terminal whenever there is a reduction. |
| int to\_int(const string&) | converts a string number to an integer and returns it. |
| int terminal(const string&) | returns the appropriate column of a terminal. |
| void copy(symbol) | copies the symbol table generated from the Lexer. |
| bool driver(symbol) | returns a boolean value if the string is accepted. The while statement will continue until an error occurred or the TOS is “ACCT.” Otherwise, it will index the rtable by converting the TOS to a numeric value representing the row and the current input token to a numeric value representing the column. After, it will determine if the first character in the indexed string is an S or R. Otherwise, if the indexed string is “ACCT,” it will end the processing and return true. If not, then the program will output an error message to the console and end the processing. If an ‘S’ is read, it will shift; meaning, we are going to push the current input token on the stack and the numeric number that proceeds the ‘S’ character. Lastly, it will pop as a means to increment the input pointer. If an ‘R’ is read, it will reduce; meaning, we are going to pop two times the number of symbols on the RHS of the numeric value proceeding the ‘R’ character. It will temporarily assign TOS to the top of the stack. Then, it will push the LHS terminal and index the rtable with the TOS and LHS terminal as numeric values. This process continues until the string has been accepted or an error occurred during processing. |

***Symbol Table:***

|  |  |  |
| --- | --- | --- |
| **Variables** | **Types** | **Definitions** |
| Instr | structure | holds three public data members: addr, op\_code, and mem. |
| scope\_res | vector<string> | holds three string values to determine the type of scope from a numeric value. |
| backup | Token | holds a Token value referencing the declaration of a token. It is used whenever a variable is declared more than once. It uses the line number of that variable to prompt the user where the previous declaration statement is. |
| list | queue<Token> | a structure that holds all information, Token, of every valid declaration of an identifier. |
| code | vector<Instr> | a structure that hold all information, Instr, of every instruction. |
| sfile | string | holds the current file name being read from the parser. |
| mLoc  aLoc | size\_t | holds the current memory/address value throughout the program. mLoc is initialized to 5000, and aLoc is initialized to 1 when the default constructor is called. |

|  |  |
| --- | --- |
| **Functions** | **Definitions** |
| bool error() | returns a boolean value if the error handler has been invoked. These types of errors are purely semantic, which consist of type, scope, and declaration errors. |
| bool exist(symbol, const string&) | returns a boolean value if the current token has been declared or not. |
| bool inScope(const string&, const size\_t&) | returns a boolean value whether the scope value from the token’s declaration is less than or equal to the current scope value. Briefly, global scopes will be more versatile since they have the lowest scope value indifference to nested scopes with higher scope values. |
| string inType(const string&, string, symbol) | returns a type value if two variable types match in assignment, conditional, and expression statements. The first check is whether an identifier or number is being read. If the token is an identifier, it will proceed to check if the SAVE\_TYPE is empty, meaning the parser is reading the beginning of those statements. Otherwise, it will check if the SAVE\_TYPE, the type of the first identifier read by those statements, matches the current token type. For example, an assignment statement can be: X = Y + Z. X’s type value will be stored in the SAVE\_TYPE variable. The inType function will check if Y and Z are the same type as X. If not, it will generate an error corresponding to an invalid type. If a numeric value is being read, it will check if the number is an integer or real. If it is an integer, it will check if the SAVE\_TYPE is a boolean value, meaning a boolean value is being assigned. Since my grammar only allows for boolean to have values of zeros and ones, it will generate an error if a boolean variable is assigned to any other value. If the SAVE\_TYPE is not an int, it will generate an error because the two type values do not match. For real numbers, if the SAVE\_TYPE is not a floating data type, it will generate an error. |
| string getScope(const size\_t&) | returns a scope reference value either GLOBAL, INNER, or NESTED. If the scope value directly corresponds to the index of the scope\_res array. Thus, GLOBAL = 0, INNER = 1 and NESTED = 2. However, any scope greater or equal to two will still be considered in a NESTED construct. |
| string getType(const string&) | returns a type value from the declaration given a token input. |
| void getInstr(const string&, const size\_t&) | stores a Instr type of the current aLoc, instruction, mLoc. |
| void insert(const symbol&, const size\_t&) | inserts into the symbol table if the current token type is not empty, meaning it is in a declaration statement, and the token does not exist in the table yet. If it does, it will generate an error corresponding to an invalid declaration. If the type is empty, meaning the parser is currently in an assignment and/or expression statements, it will check if the variable has been declared, meaning if it exists in the table. If it does not exist in the table, the symbol table will generate an error corresponding to an undefined token. Furthermore, it also must be within scope. If it is not within the scope, the symbol table will generate an error corresponding to an out-of-scope reference. |
| void insert(const size\_t&, const string&, const string&) | stores a Token type in a symbol table from the Lexer that will be used in the Syntactical phase in order to parse a string and determine whether it is syntactically correct or not. |
| void sprint()  void aprint() | prints all information of each element, either Token or Instr, to the console. |
| Token find(symbol, const string&) | returns a Token type if the input token exists in the table. This function is used to obtain a token’s memory address to store in the instruction table. |
| Instr& find(const size\_t&) | returns an Instr type reference. This function is used in the backPatch function, where this function is called to modify or replace the memory address of the SAVE\_ADDR from the jump\_stack. |

***Error Table:***

|  |  |  |
| --- | --- | --- |
| **Variables** | **Types** | **Definitions** |
| Token | structure | holds five public data members: ln, tok, lex, mem, type, s\_ref. |
| e | queue<string> | holds all errors generated from the Lexical, Syntactical, and/or Symbol Table. |
| efile | string | holds the current filename of the file where an error occurred. |

|  |  |
| --- | --- |
| **Functions** | **Definitions** |
| void handler(symbol, const string&, const size\_t&, const size\_t&) | pushes errors in response to the Symbol Table. The table parameter will be used to determine what token on what line number the error occurred in. The file name, sfile, is passed to indicate what file the error occurred in. The backup token is also passed and is only used to reference the previously declared variable’s line number. The last parameter, i, is used to determine what error type from the switch case statement. |
| void handler(const size\_t&, const string&) | pushes errors in response to the Lexer. It will record the efile and determine which error type given an invalid token. If the first character in the input string is a numeric value, it is enough information to generate an error since it ended in an invalid state. Otherwise, if a period symbol is found, then there are at least two period symbols in the string. If the first character is not a numeric value, it must be either a dollar or underscore symbol. |
| void handler(const Token&, const string&) | pushes errors in response to the PRDP. Following the actions of pushing the efile, it will find the correct error type given the string, q. A Token, p, structure is also passed as a reference to what value on what line number the error occurred. |
| void display() | outputs all errors to the console. |
| bool empty() | returns a boolean value if an error occurred or not. |

***Parse Tree:***

***Node Structure:***

|  |  |  |
| --- | --- | --- |
| **Public Data Members** | **Types** | **Definitions** |
| value | T | holds the symbol, either terminals or nonterminals as the current parent node |
| parent | Node<T>\* | holds a pointer to a symbols’ left-hand side nonterminal |
| child | vector<Node<T>\*> | holds an array node pointer from the current node |

|  |  |
| --- | --- |
| **Functions** | **Definitions** |
| Node(T v) | sets ‘v’ to the current node’s value, parent node pointer to nullptr, and child array node pointer to nullptr. |
| Node(T v, vector<T>) | sets ‘v’ to the current node’s value, parent node pointer to nullptr. Lastly, it will iterate through every element in the array and create a new node from the Node(T v) function. Those new nodes will be indexed into the current node pointer’s child pointer. |

***Tree Class:***

|  |  |  |
| --- | --- | --- |
| **Private Data Members** | **Types** | **Definitions** |
| root | T | holds the symbol, either terminals or nonterminals as the current parent node |

|  |  |
| --- | --- |
| **Public Data Members** | **Definitions** |
| Tree() | holds the symbol, either terminals or nonterminals, as the current parent node |
| Node<T>\* getRoot() | returns the root node |
| T printTree() | returns a string consisting of a preOrderTraversal. Additionally, you may change the type of traversal to BFS (Breadth-First Search) traversal; however, it is not completely implemented. |
| T BFS(Node<T>\*) | a Breadth-First Search traversal of the tree. A node iterator points at the first element in the child node array. It will iterate until the iterator reaches the last element in the array. During then, the function will first remove the ~ symbols to make the output look cleaner. Next, it will output to the console the current node’s value. After every child node’s value has been printed, it will proceed in a pre-order fashion. However, the function will check if the next child in the pre-order traversal is empty or not. If it is empty, we have reached a terminal since every terminal are leaves in the tree. If not empty, it will call recursively call the function while incrementing the index array value, j, which is defined statically. |
| T preOrderTraversal(Node<T>\*) | The function will first print out the current node’s value and proceed left in the pre-order traversal. After a nullptr has been reached, the function will exit on the recursive call and proceed to the next child of the current node. In my implementation, it first removes the ~ symbols to make the output look cleaner. It will proceed to print the parent node and current node’s value. The for loop will iterate starting from the root’s first child to the root’s last child. It will also have a base case for when the next child node points is empty, nullptr. If the next child node is not empty, then my function will recursively call the next child node from left to right. |
| void makeTree(T, const vector<T>&) | inserts a production rule into the tree by checking if two nonterminals match. If they do, the child node will be replaced by the new node. In addition, a ~ symbol is inserted at the end of each inserted node to conclude that the nonterminal has already been expanded or processed. |
| void makeTreeRecursive(Node<T>\*, Node<T>\*, Node<T>\*) | this function will insert a new node into the tree. It will traverse the tree from left to right until the new node’s value matches with an existing node’s value. If it does, the function will append a ~ symbol indicating this symbol has already been processed. The base case is when the next child node is empty, nullptr. It will exit and proceed with recursion. |

**3. Limitations**

My grammar only accepts single relational operators since my Lexer cannot identify dual operators.

**4. Shortcomings**

None