Course: CPSC 323

Assignment #3 Documentation

Name : Jason Duong

Date:

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 different approach than the one used for the last project. In addition, students must also implement symbol table handling and type checking or a parse tree implementation. More points will include 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

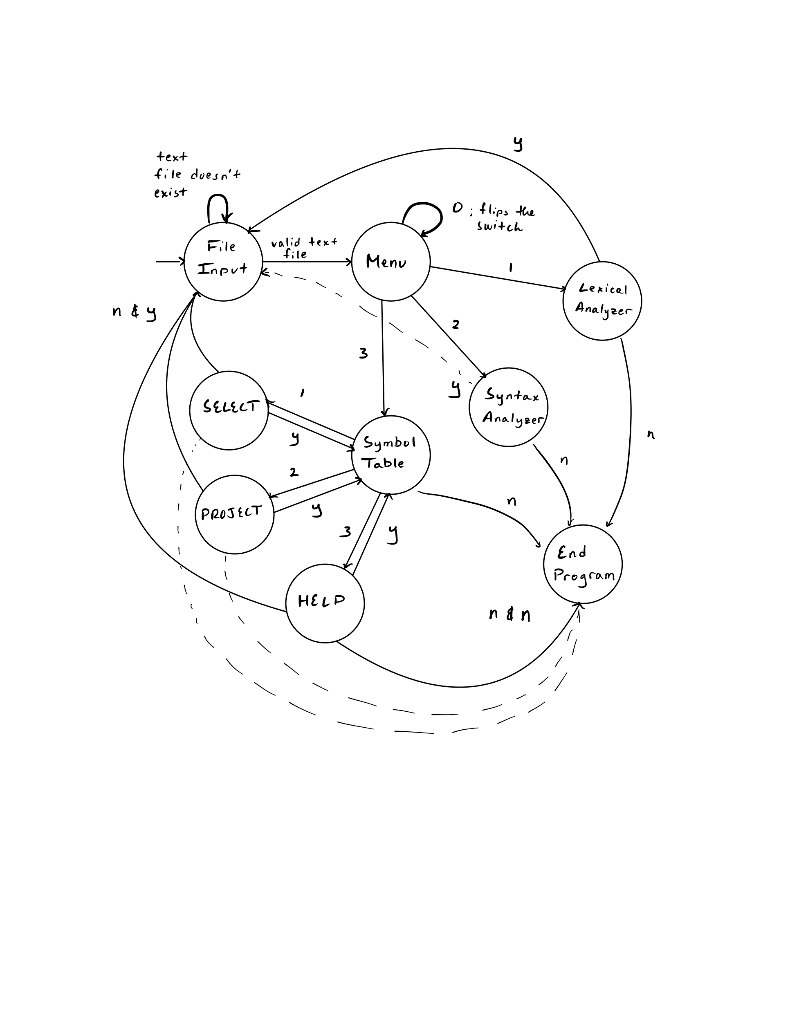
| └──────── **main.cpp**

| └──────── **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.

1. **Design of the program**

Console Navigation State Diagram for PRDP:



*LEXER:*

Added

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

save

jump\_stack

backpatch()

void printAL() – prints all assembly instructions (intermediate code generation)

void getTree() – prints a parse tree after concluding the statement is syntactically correct

Node<string>\* getRoot() – returns the root of the stored tree

makeTree(T, const vector<T>&) –

1. In “tree.h”:

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.

1. In “PRDP.h”

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.

CURR\_TYPE

SAVE\_TYPE

CURR\_SCOPE – 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, then it will decrement the scope by one. In addition, when ever an identifier is called, the current scope value will be inserted into the symbol table. Of course, the symbol table will check if the identifier was not a double declaration or any other invalid use before inserting the current scope value.

CURR\_ADDR

OP

*Stack:*

My stack will push the end of string character “%” and the starting symbol “S” when the constructor is called. If the top of the stack is an terminal symbol, my stack will validate whether it matches with the input string. Otherwise, it will pop the stack and find a production rule from the pre-defined predictive table. The functions NONTERMINAL() and TERMINAL() will generate a numeric value corresponding to the rows and column of the predictive table. If there is a match, cell is not equal to zero, then it will push the push the right-hand side productions in reverse order since the stack data structure is LIFO. If a particular rule corresponds to a epsilon, my program will only pop the stack and not push anything since the string will be accepted for that rule if there is an absence of input.

Lastly, I have added a parse tree implementation using a tree data structure. Instead of a conventional binary tree where each parent node points to two child nodes, my tree allows for more than two children by utilizing a vector<Node\*> as a list of pointers. The insert() function will traverse the existing tree and will try to find a match with the nonterminal symbols. It will also skip pass terminal symbols when necessary.

*LR:*

Symbol Table:

I have added two new functionalities to the symbol table, SELECT and PROJECT. The select function will find matching tokens, lexemes, and line numbers given three inputs. A space should follow the three inputs. The user can input a ‘\*’ to represent all data from that column. If a user wants a specific token, lexeme, or line number, they would need to enter the specific value. A valid input for the select function could be SELECT \* a 5. This will filter all lexemes that are valued ‘a’ and are on line 5. However, there are no restrictions to the token column. This represents an intersection operation since we are narrowing down on a specific row. My query does not work for union operations where users can select more than one value per column. For example, the union operation could be like SELECT \* a U b \*. This will ultimately find all lexemes, which are ‘a' or ‘b’ of any token and line number and output it.

The project function allows users to eliminate certain columns in the symbol table. There are three Input values, like the select function. However, the inputs are either ‘y’ or ‘n’. A ‘y’ input means that the column will be displayed. An ‘n’ input means that the column will not be displayed. For example, a user can input PROJECT y n y. This will only output the tokens and line numbers column, eliminating the lexeme column.

Lastly, my program for the symbol table does not integrate the select and project functions together. What I mean by that is if a user wants to find a specific lexeme and only wants to display the line numbers column.

Error Table:

There is one main function in the error table file, the handler. The handler function is overloaded. One is for the lexical errors, errors due to the misspelling of lexemes, and the other is for the syntactic errors, errors due to the invalid ordering of tokens. The lexical handler is already discussed in the documentation in assignment 1. The syntactic handler will receive three values for its argument: line number, expected lexeme, and proceeding lexeme. The non-terminal will be used to determine the type of error. After, it will add the line number to where the error occurred in the code and print the expected lexeme. My error handler will also record the error token position by prompting the user to what lexeme it is after.

1. **Limitations**

None

1. **Shortcomings**

None