Course: CPSC 323

Assignment #2: Syntax Analyzer Documentation

Name : Jason Duong

Date: Tuesday, April 13, 2021

1. **Problem Statement**

The second assignment is to write a syntax analyzer based on the strategies and grammar provided in class and implement it using either an RDP (recursive descent parser), PRDP (predictive recursive descent parser), or a Stack (table-driven predictive parser) for maximum points.

***Please grade the PRDP parser.***

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/Syntax/PRDP

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., ‘2.txt’

b. Using the Executable File

Extract the folder: Compiler

Double click on the extracted folder and locate to the PRDP folder

Double click on the executable file: main.exe

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

└─────────── **PRDP**

| └──────── **test**

| └──────── **1.txt**

| └──────── **2.txt**

| └──────── **3.txt**

| └──────── **4.txt**

| └──────── **error1.txt**

| └──────── **error2.txt**

| └──────── **PRDP.h**

| └──────── **main.cpp**

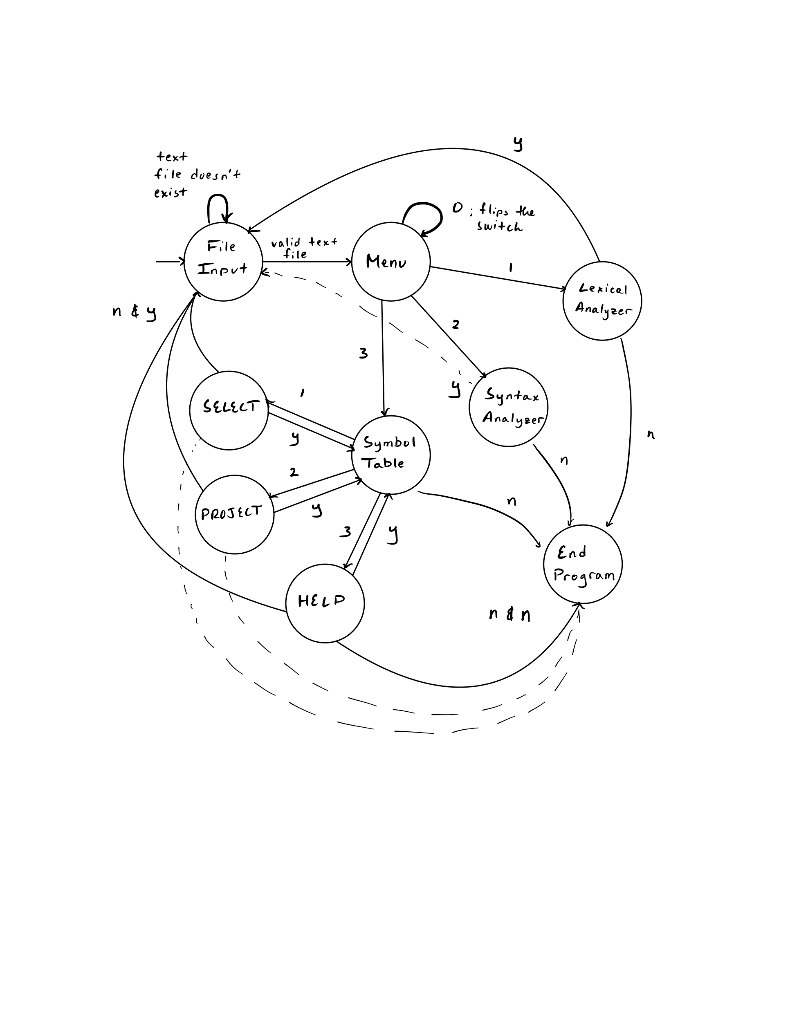
| └──────── **main.exe**

| └──────── **out.txt**

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:

Updated to hold line numbers in every token

Added default constructor and destructor

Added function insert() which only inserts into the symbol table rather than printing

Functionality and Design is same according to assignment’s 1 documentation

PRDP:

***( The end of character or string ‘%’ is used because my lexer does not accept ‘$’ as a valid string )***

G = (T, N, S, R)

T = { ϵ ; , bool int float = + - \* / ( ) id integer real if then else endif while do whileend begin end < <= == <> >= > }

N = { <S>, <A>, <D>, <E>, <IF>, <WHILE>, <BEGIN>, <TY>, <ID>, <MID>,

<T>, <Q>, <F>, <R>, <NUM>, <C>, <B>, <MS>, <RELOP> }

S = <S>

R = {

<S> -> <A> | <D> | <E> | <IF> | <WHILE> | <BEGIN>

<D> -> <TY> <ID> <MID> ; | <empty>

<MID> -> , <ID> <MID> | <empty>

<TY> -> bool | int | float

<A> -> <ID> = <E> ;

<E> -> <T> <Q>

<Q> -> + <T> <Q> | - <T> <Q> | <empty>

<T> -> <F> <R>

<F> -> ( <E> ) | <ID> | <NUM>

<R> -> \* <F> <R> | / <F> <R> | <empty>

<ID> -> id

<NUM> -> integer | real

<IF> -> if <C> then <S> else <S> endif

<WHILE> -> while <C> do <S> whileend

<BEGIN> -> begin <S> <MS> end

<MS> -> ; <S> <MS> | <empty>

<C> -> <E> <B>

<B> -> <RELOP> <E> | <empty>

<RELOP> -> < | <= | == | <> | >= | >

}

**First Set:**

First (<S>) = { id, bool, int, float, (, integer, real, if, while, begin }

First (<A>) = { id }

First (<D>) = { bool, int, float, ϵ }

First (<TY>) = { bool, int, float }

First (<MID>) = { , ϵ }

First (<E>) = { (, id, integer, real }

First (<Q>) = { +, -, ϵ }

First (<T>) = { (, id, integer, real }

First (<F>) = { (, id, integer, real }

First (<R>) = { \*, /, ϵ }

First (<ID>) = { id }

First (<NUM>) = { integer, real }

First (<IF>) = { if }

First (<WHILE>) = { while }

First (<BEGIN>) = { begin }

First (<C>) = { (, id, integer, real }

First (<B>) = { <, <=, ==, <>, >=, >, ϵ }

First (<MS>) = { ; ϵ }

First (<RELOP>) = { <, <=, ==, <>, >=, > }

**Follow Sets:**

Follow (<S>) = { % else endif whileend ; end }

Follow (<D>) = { % else endif whileend ; end }

Follow (<MID>) = { ; }

Follow (<TY>) = { id }

Follow (<A>) = { % else endif whileend ; end }

Follow (<E>) = { % else endif whileend ; end ) < <= == <> >= > then do }

Follow (<Q>) = { % else endif whileend ; end ) < <= == <> >= > then do }

Follow (<T>) = { % else endif whileend ; end ) < <= == <> >= > then do + - }

Follow (<F>) = { % else endif whileend ; end ) < <= == <> >= > then do + - \* / }

Follow (<R>) = { % else endif whileend ; end ) < <= == <> >= > then do + - }

Follow (<ID>) = { % else endif whileend ; end ) < <= == <> >= > then do + - \* / = , }

Follow (<NUM>) = { % else endif whileend ; end ) < <= == <> >= > then do + - \* / }

Follow (<IF>) = { % else endif whileend ; end }

Follow (<WHILE>) = { % else endif whileend ; end }

Follow (<BEGIN>) = { % else endif whileend ; end }

Follow (<MS>) = { end }

Follow (<C>) = { then, do }

Follow (<B>) = { then, do }

Follow (<RELOP>) = { % else endif whileend ; end ) < <= == <> >= > then do }

My parser is implemented using a predictive recursive descent parser. I have also implemented using the stack approach but would prefer the PRDP to be graded. For the PRDP, it uses a queue<Token> data structure where token is a struct defined in the symbol table. My parser will read the entire code and generate any errors that occurs. I also used four #define statements to shorten commonly used code. Every non-terminal function in my program will match this structure: for every non-terminal reached, check the first set, print the tokens and lexemes, print the production rules for the current token by the print() function, call the lexer() function. If the first set fails, it will check the follow set only if that production rule can take in an absence of input. The lexer function will record the current token and then get the next token by popping from the queue. In addition, the lexer function also backups the current token for reference if an error occurs. The print(NONTERMINAL) and print(TERMINAL) functions will print the production rules given an enumerated typed value. In this way, it can be easily understood what production rules are going to be called. It would be more tedious to reference the arguments as numbers.

Whenever a grammar is deemed syntactically correct, it will output to the file as \*\*\*ACCEPTED\*\*\* along with all the lexemes following it. Otherwise, my program will output to the file as \*\*\*NOT ACCEPTED\*\*\* and generate an error message to the console with the appropriate line number, expected lexeme, and after what lexeme had it occurred. After the entire text file is read, my program will output to the console ‘PASS’ if no errors had occurred. It checks if the queue data structure for the error table is empty. Lastly, the next\_lexeme(symbol) function is used only for the assignment production. Since if the first lexeme is an identifier, it can be either an expression or assignment statement. If I use the lexer() function, it will affect the queue since it was passed by reference. Thus, the next\_lexeme function is passed by value and will not affect the queue. Therefore, my program will check the next lexeme for an equal sign and proceed if it is. Otherwise, it will start processing as an expression statement.

Stack Implementation (Do not grade):

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.

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