**CS323 Documentation – Assignment 2**

**1. Problem Statement**

Write a syntax analyzer (parser). Build the entire parser using a top-down such as an RDP, a predictive recursive descent parser or a table driven predictive parser.

**The Parser:**

Write a procedure (Function) – lexer (), that returns a token when it is needed. Your lexer() should return a record, one field for the token and another field the actual "value" of the token (lexeme), i.e. the instance of a token.

The main program should test the lexer, reading in a file containing the source code given from class to generate tokens and write out the results to an output file. Both tokens and lexemes must be printed.

**2. How to use your program**

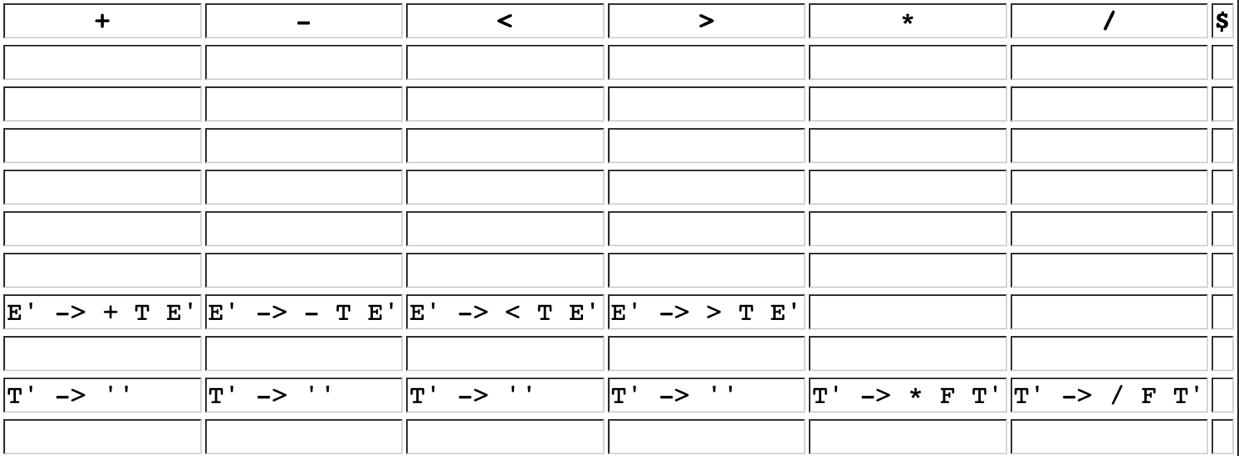
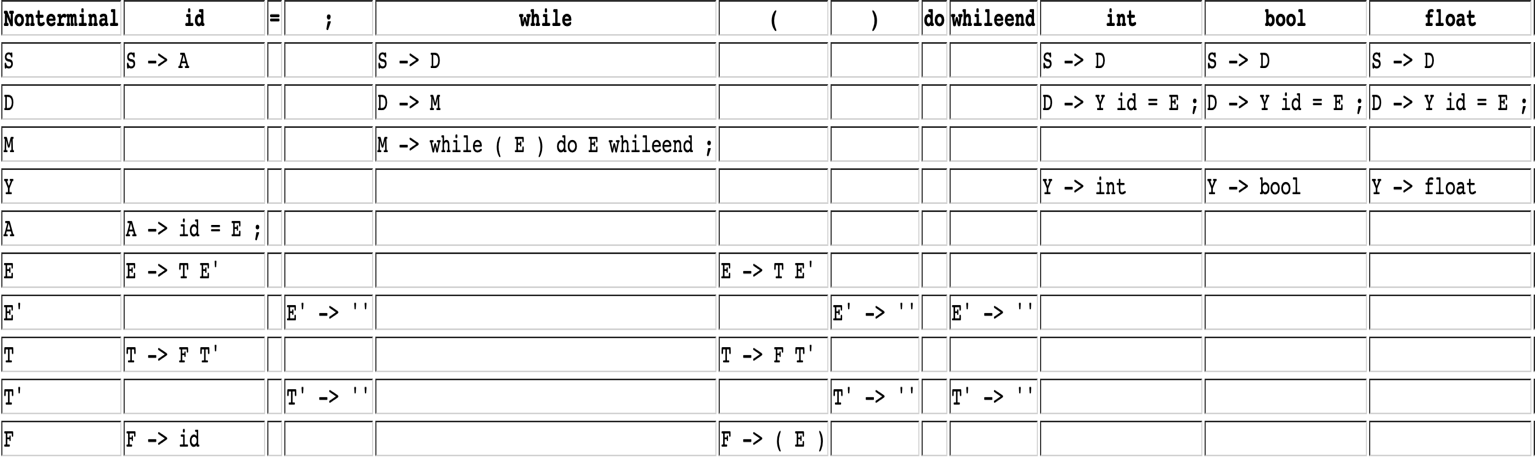
Running the program can be accomplished in MacOS or Linux by running the bash file:

1. Open the terminal in the main file titled “Assignment2\_JustinDrouin”
2. In the terminal type **bash RunLex.sh for Linux** or **Mac type sh RunLex.sh**
3. The terminal will ask for a source file. Enter the complete text file name with extension. ( input.txt )

**3. Design of your program**

The parser is designed using a LL(1) Table – table predictive parser. The parser is located in the token class as well and the outfile is named “tokenOutput.txt”. The program executes through a bash file that accepts a .txt file name for the source code. The table is designed using a map container and utilizes enumeration. If the syntax analyzer defaults it prints and error code with the line number and also prints it in the file. The table is laid out as such:

|  |
| --- |
|  |



**‘ ‘ =** ε

**Grammar:**

S -> A | D

D -> Y id = E ; | M

A -> id = E ;

M -> while ( E ) do E whileend ;

Y -> int | bool | float

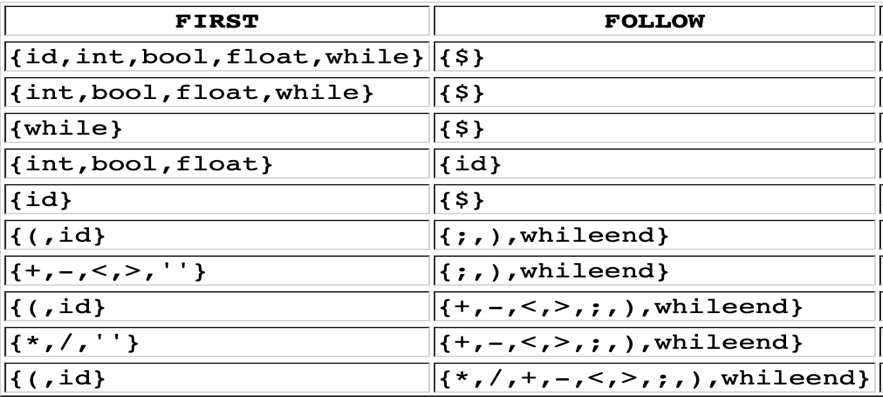
E -> TE’

E’ -> +TE’ | -TE’ | <TE’ | >TE’ | ε

T -> FT’

T’ -> \*FT’ | /FT’ | ε

F -> (E) | id



**4. Any Limitation**

The main limitation is with the lexer itself. The parser works exactly as intended, but with the way the lexer is designed it does not work as harmoniously with the parser as I would like. The type must also be initialized ( int x = b ; ). The main issue around this assignment has mainly been around making the lexer work with the parser. It can also run keywords “while”, “do”, and “whileend”. The syntax for this must be “while ( expression ) do expression whileend;”. The other limitation of it is that it works best on single line statements or statements that are all on one line, and it can get buggy if spread across multiple. The while statement also flags ‘>’ in the condition as an error and returns a syntax error code. This is due to the grammar for while. It however shouldn’t not be flagging as an error and will require further testing.

**5. Any shortcomings**

I did get to implement the third iteration- the if-while grammar, but not as much of it as I would have liked. I also would have liked to implement a better error system, it can be improved.