**CS323 Documentation**

**Problem Statement**

The program is a syntax analyzer. It involves either using the top-down approach such as RDP,a predictive descent parser or a table driven predictive parser approach. Then, use the lexer() generated in assignment 1 to get tokens. The parser should print to an output file. The output file should contain the tokens, lexemes and production rules used. Finally, error handling must report syntax errors that occur. You may exit the program when the first error is reported. The error message should be meaningful (i.e. message with token, lexeme, line number and error type).

**How to use your program**

The program has only been tested on the Linux operating system running the newest version of Python3.9, any incompatibility will be due to the user not using the correct version of python. The program requires a python version greater than 2.5 due to the use of ternary operators.

The data.txt file, which contains the input script, must be in the same directory as the Syntax\_Analyzer.py python script. To run the program, enter the directory of the data file and python script and type: $ python3 Syntax\_Analyzer.py

Once the program is opened, it will prompt the user for input:

‘Enter file you would like to open (type “quit” to exit): ’

You may enter the name of a file you would like to input such as: testcase1.txt

Once a user inputs a file, the program executes. Once the syntax analyzer is finished, the program prompts the user whether they would like to process another file. ‘yes’ input will ask the user for another input file, ‘no’ input will exit the program.

Once the program quits, the files of the lexer contents and the syntax analyzer are saved to the working directory as ‘data.RAT’ and ‘data.SA’ respectively.

If a file is not recognized the program will respond with:

‘Your file was not found!’

Then it will prompt the user ‘Would you like to process another file? (yes/no):’

‘no’ entry will exit the program. ‘yes’ will ask the user for another data file input.

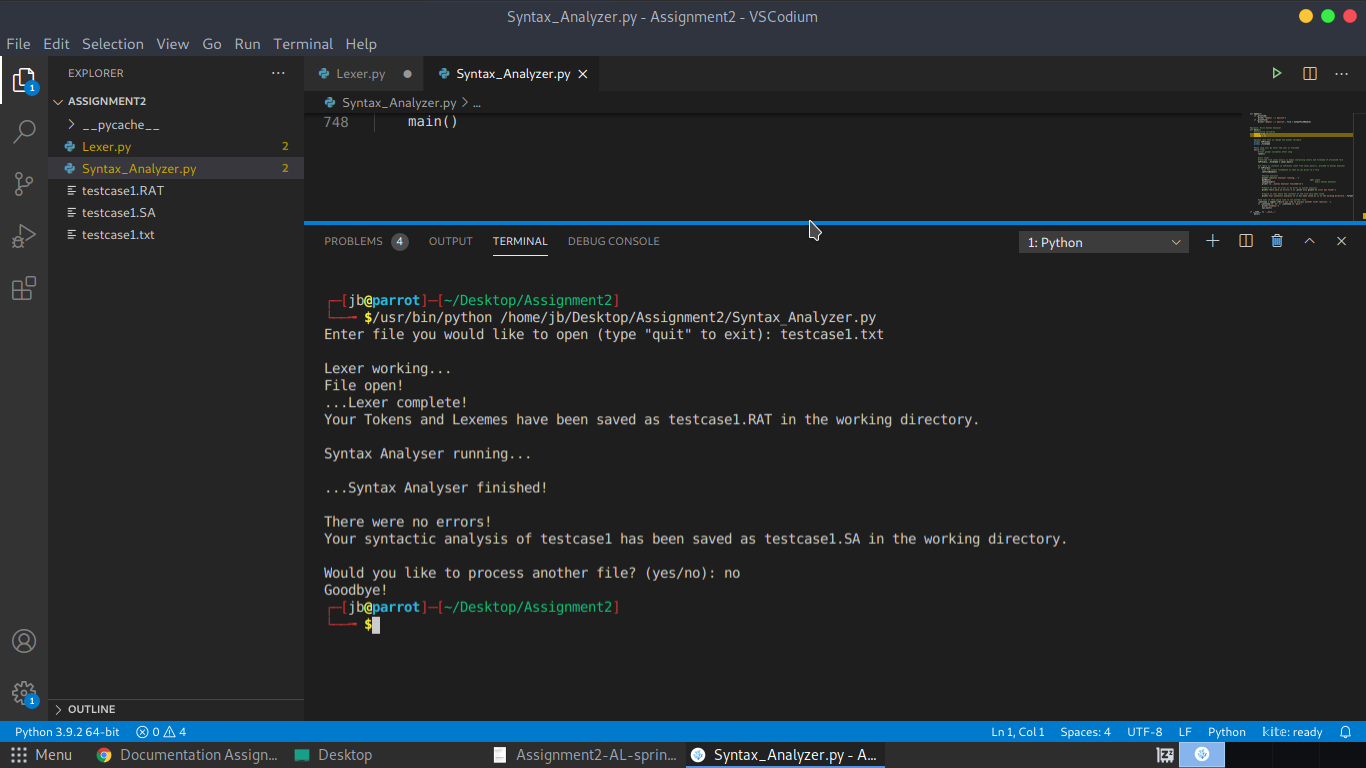
To quit the program type: quit into the terminal and the program will exit.

Successful runs will output to the terminal and be written into a file with the name of the input (user defined) file and the extension “.RAT”.

Once the program is called, it will run on a loop until the user enters: quit

If the syntax analysis has an error, the error is reported to the terminal as well as stored in a file.

Example run:



**Design of your program**

The program ‘Syntax\_Analyzer.py’ has 39 functions, 32 production functions and 7 helper functions. A RDP style parser is used. Initially the ‘main()’ function is called. The ‘main()’ function calls the Lexer.py program created in assignment 1. The lexer returns a deque of Lex objects and a filename from the input file (i.e. if file is data.txt it returns data). The first Lex object is set to the ‘current’ global Lex object via the getNext() function and then rat15S() is called. If the file is syntactically correct, the program continues until the file is finished and then output is stored in a data.SA file in the working directory (data is the filename from user’s input). If an error occurs in the syntax, it is reported to the screen and the output is saved in a data.SA file in the working directory.

For the second assignment, the Lexer.py program was redesigned to include a Lex() class. This is so that one object, a Lex, could contain the information for the token, lexeme, and line number. This was stored in a deque and passed into the syntax analyser. The syntax analyzer was built using a RDP. Every production rule has its own function, which decides on the path for the token. There were a few cases of left recursion that had to be solved before implementation.

Left recursion occurred with the following production rules:

<Expression> ::= <Expression> + <Term> | <Expression> - <Term> | <Term>

<Term> ::= <Term> \* <Factor> | <Term> / <Factor> | <Factor>

Solving for Expression:

<Expression> ::= <Term> <ExpressionPrime>

<ExpressionPrime> ::= + <Term> <ExpressionPrime> | - <Term> <ExpressionPrime> | <empty>

Solving for Term:

<Term> ::= <Factor> <TermPrime>

<TermPrime> ::= \* <Factor> <TermPrime> | / <Factor> <TermPrime> | <empty>

The solutions to these were implemented in the code and the function names are the same as seen above. Implementation of which is found in the code under the corresponding names for the functions.

An error() function was created so that the production functions could pass the expected lexeme into error() and the function would report the expected lexeme as well as what the current lexeme/token/line number were. This was done so that there was no need to generate a separate error message for every function.

**Any Limitation**

The line numbers are incorrect if there are comments in the code. The comments are ignored and code is still parsed correctly however while ignoring the commented code the lines are ignored as well.

**Any shortcomings**

None