Kennesaw State University

Department of Computer Science

4308, Concepts of Programming Languages, Section 01

Parser Submission Report

George Tatge: [gtatge@students.kennesaw.edu](mailto:gtatge@students.kennesaw.edu)

10/19/2020

**Initial Problem Statement**

The goal of this part of the project is to take in our input from the scanner part of the project and pass it through a program that will create a parse tree to give structural representation of the input while also checking for correct syntax. The purpose of this report is to explain and show how we decided to solve the task of creating this parser. In this part we design the parser section for our interpreter so we can create a parse tree and check for syntax errors while we are there.

**Summary and Purpose of the Assignment**

To start off this part of the project we had to come up with a way to create the parse tree and which type of parse tree we were going to use. We looked at the different options and weighed which one we thought we would want to work with the most. We decided to use the Recursive Descent Parser because it mimics the BNF style of grammar when you write it. It is very user-friendly system, and a lot of programming languages tend to have their language written using BNF.

**Parse Tree Explanation**

We designed the parser first by creating a grammar that passed the pairwise disjointness test; we made sure no rule contained itself as the first element in its right-hand side. Next, we started creating a recursive descent parser with almost the exact same structure as our grammar. Each rule described a method, and each nonterminal token in a rule was evaluated by calling another method and completing that first. We kept track of how far into the program we had parsed using an iterator, TokenStatus, which each method modifies (or does not if it determines the sequence of tokens cannot be a rule) and then returns.

In this manner we managed to implement print statements, assignment statements, and the evaluation of string expressions. However, we ran into a problem when trying to evaluate numeric expressions: we could no longer assume that every operation would be executed from left to right, and could no longer immediately add an operator as soon as it was seen. We did not know how to modify our grammar to perform multiplication and division first, either. Instead, we decided that when evaluating a numeric expression, we would look ahead to a token that would indicate the expression had to end (a terminal) and get each number or operator in the expression. Then, we would add the numbers and operators to separate lists. Then, we would (from left to right) repeatedly iterate through the operators based on levels of precedence, and calculate the new value they return based on the numbers to their left and right, implicitly adding them to the parse tree.

**Program Design and Error Checking**

For design of the overall program, we used GitHub to help spread out the project into different areas and allow for more modularity. This allowed us to push and merge files that we are working on without getting in each other’s way or relying on each other to do get certain things done first and upload them. We created different files for different things we needed. Added separate text files to input ADA language into the program. Different .py files to break up the parser and lexer. A token status python file to add a bunch of functions to traverse tokens. As for error handling we used try and except statements to check for errors and log them. Functions to check for errors were created to be used along the way in the program when creating the parse tree. The functions take a token status and the expected value and if it does not match it will raise and exception with information on where the error has occurred. We implemented this by checking for numerous syntactical errors that can occur when writing programs. A few examples are: checking for an undeclared variable, checking for unexpected char, and checking for unexpected operand. A lot of the error checking happened along the parse tree creation by using if and else statements to check for errors each step of the way.

These are the main functions used to check for errors.

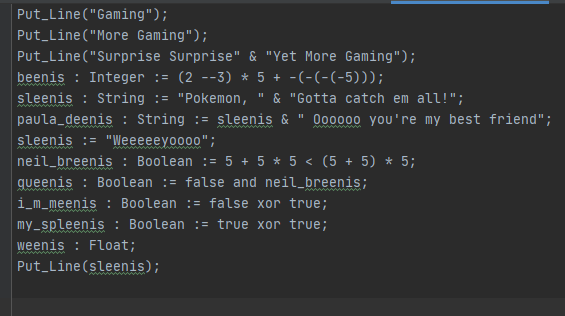


**Challenges**

Most of the challenges I have already covered along the way in this report. One small challenge was trying to figure out how to check for comparison between two different arithmetic functions. Since we would need a way to know where the function stops and not just the whole line, we concluded that using the symbol or keyword that separates the two functions could be used as an arbitrary terminal symbol or just a terminal symbol for that specific function but not the entire line.

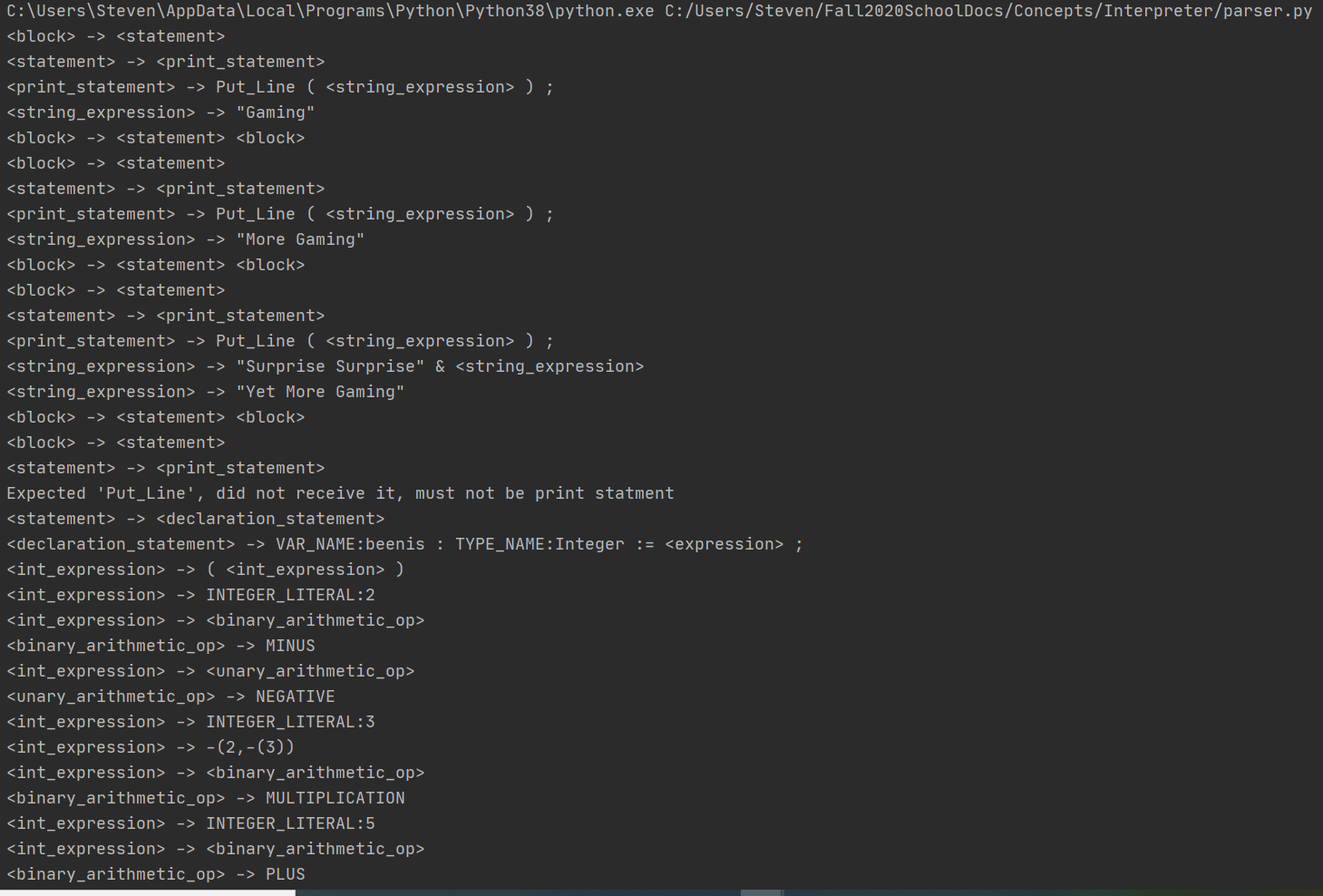
**Input Data and Results**

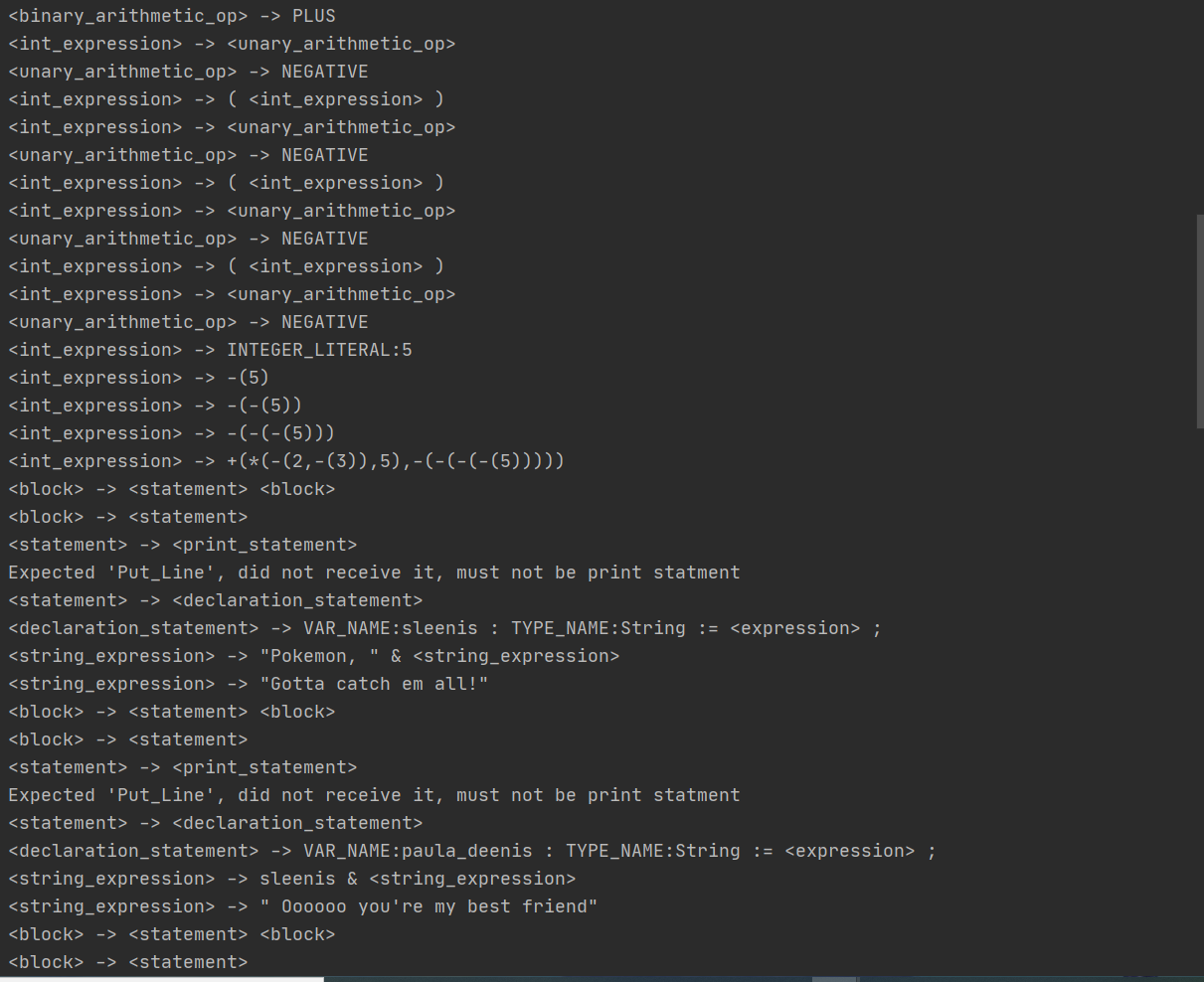
Here is what got inputted into the program

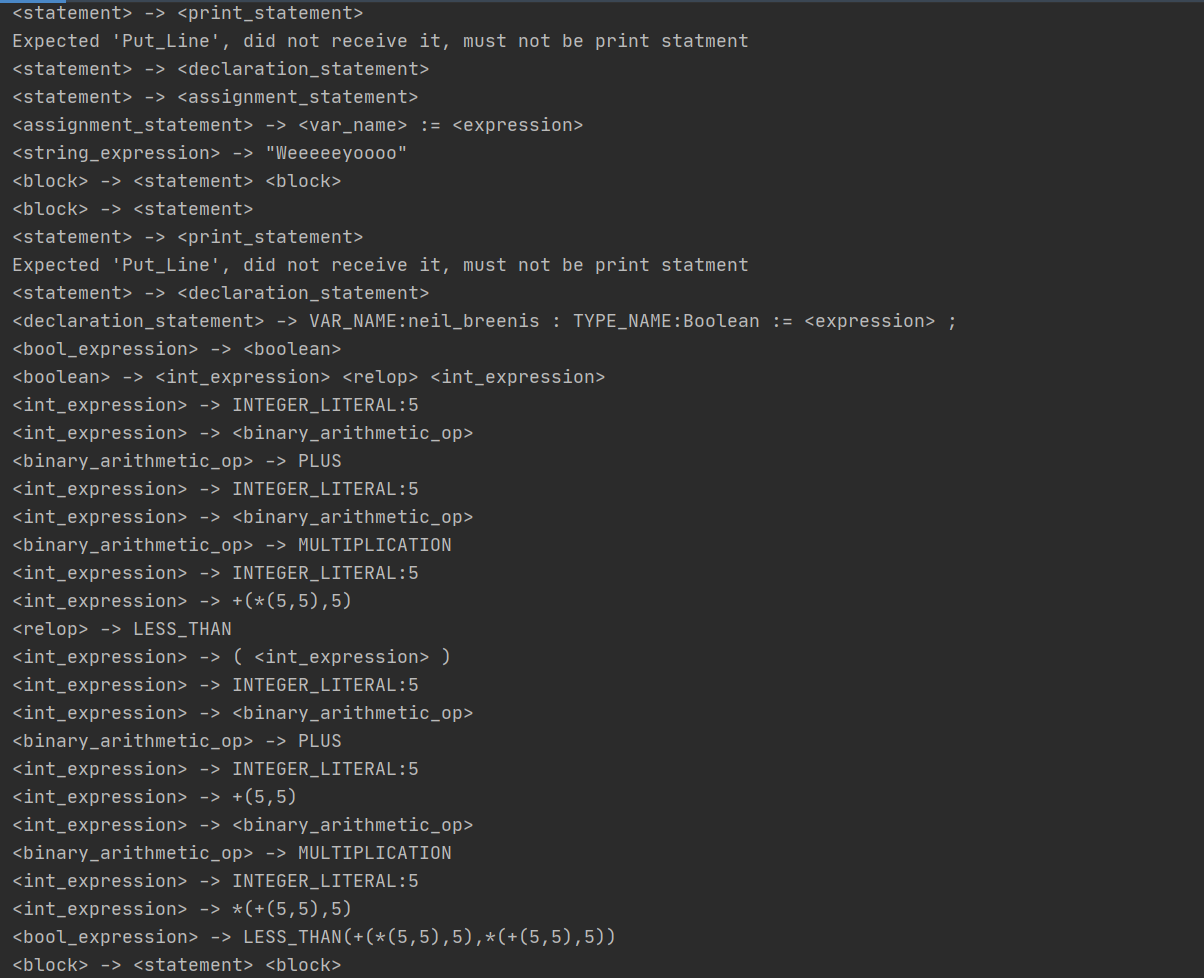


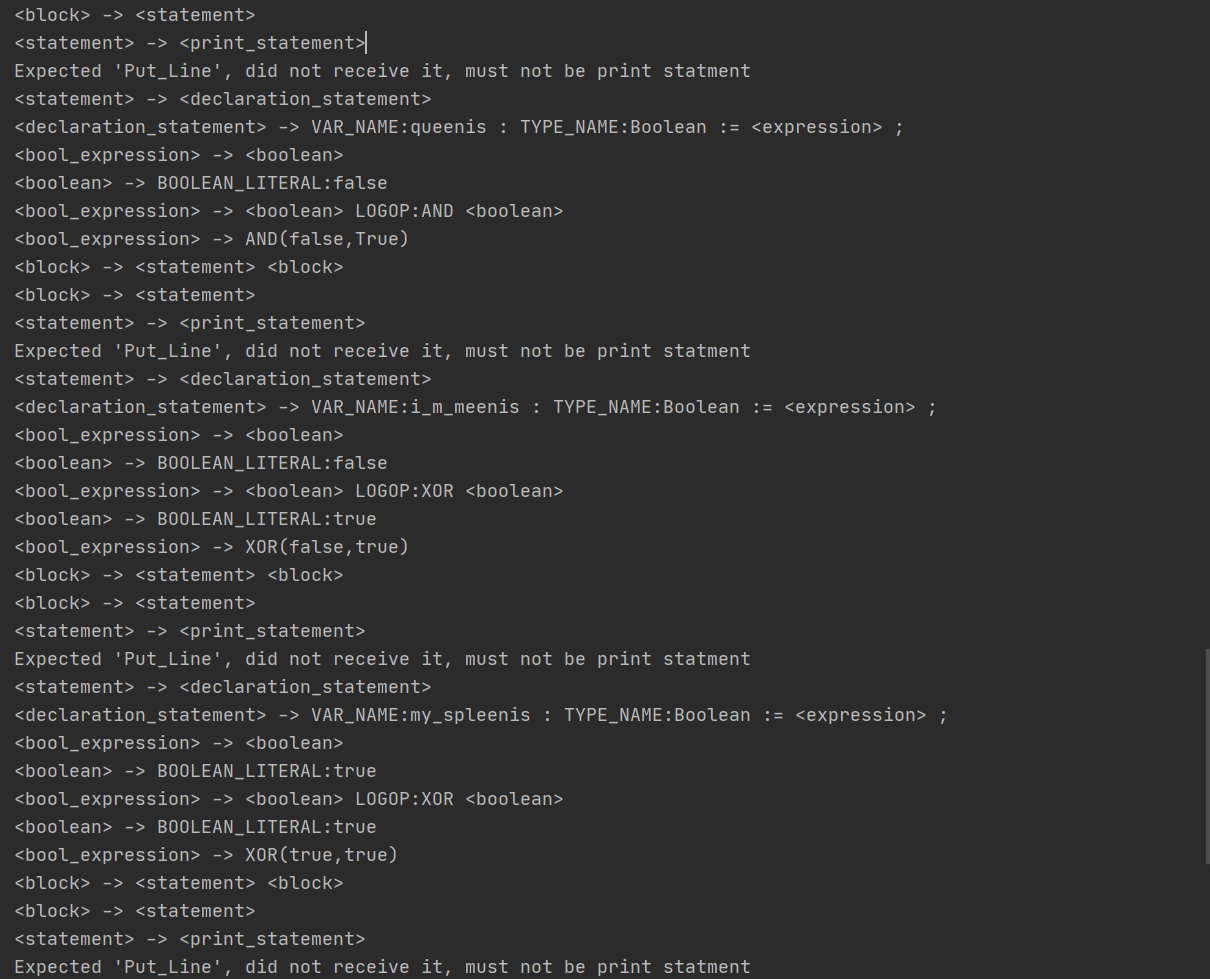
And here is the output from the inputted data

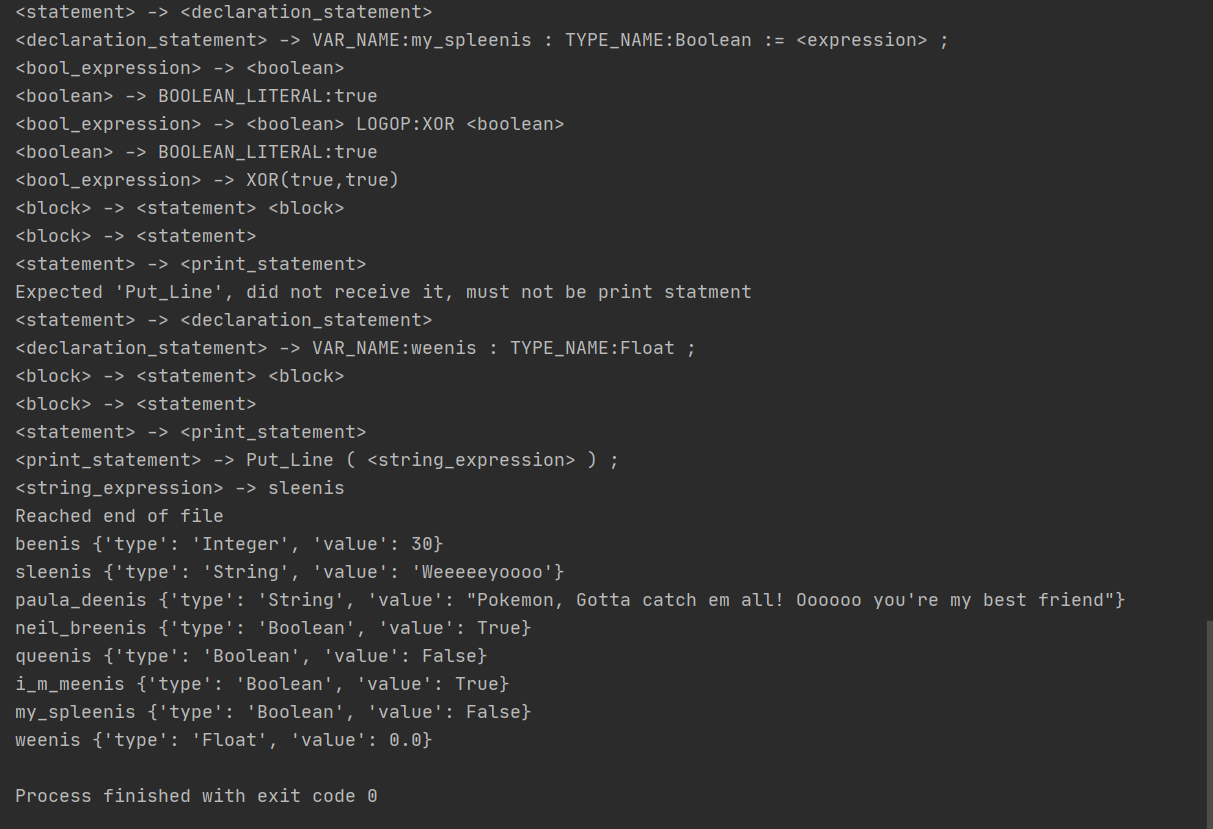
|







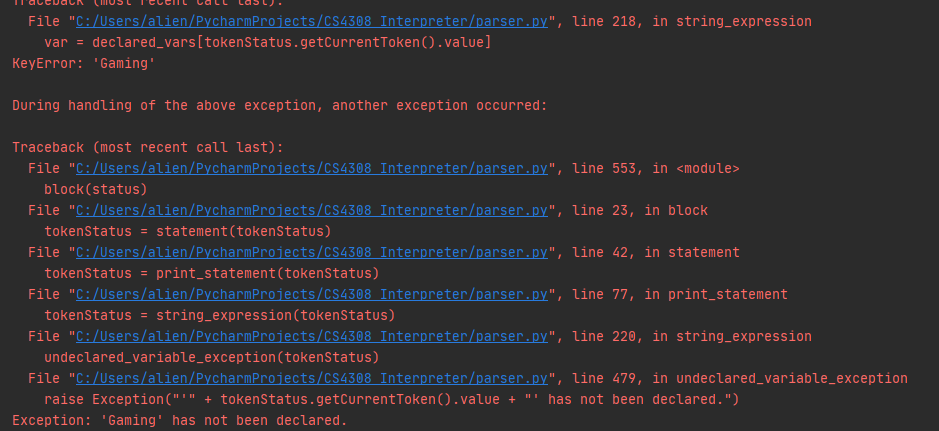




**Error Testing**

Trying to print an undeclared variable

Put\_Line(Gaming);



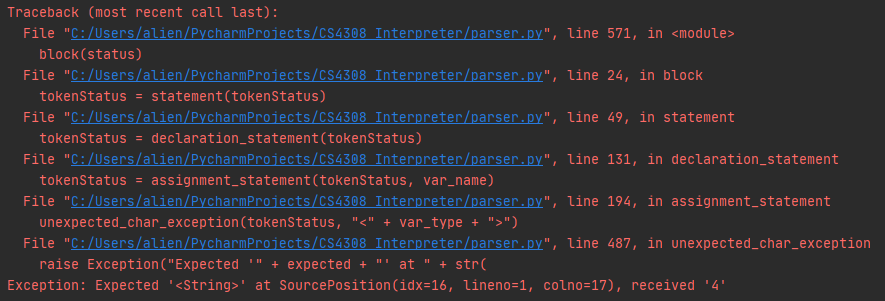
Forgetting semi colon to end line

Put\_Line("Gaming")



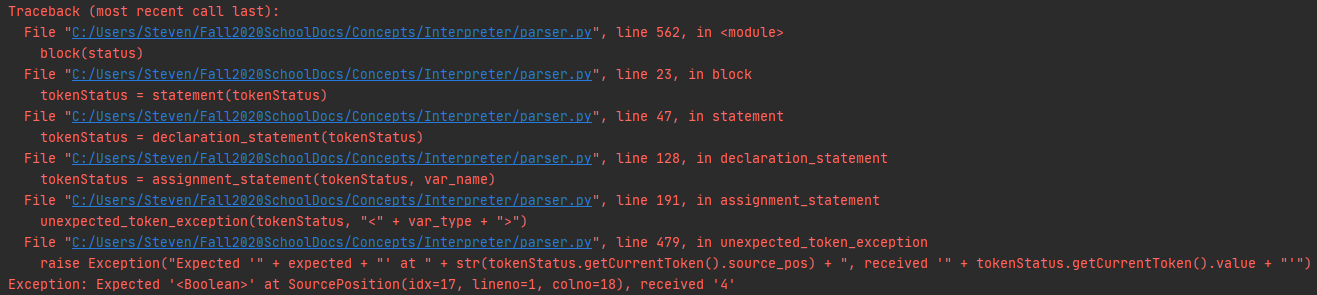
Declaring a string wrong

bob : String := 4;



Declaring a Boolean wrong

bob : Boolean := 4;



Declaring an integer wrong

bob : Integer := true;



**Limitations of Design**

A Limitation of this design is relying on python to basically interpret another programming language. It is like relying on one language to interpret another. Another limitation could be only being online to get things done which requires more structure and organization to get things done.