Project 2: Parsing SpartyTalk

DUE: Sep 30, 2022 11:59 PM EDT**. *Please make sure to always make backups of your files!***

Description

In this project, we continue implementing the language called *SpartyTalk* by *tracing the parsing of source code.* The parser we implement in this project should be based on the lexer implemented in Project 1.

First, let us do a quick recap of what *SpartyTalk* is. The following is a sample program written in *SpartyTalk*:

```
gogreen;
nvar a = -10.5;
svar b = “hello\n”;
svar c = “world”;
svar d = b + c + c;
nvar e = a * 2;
nvar f = 3.5 / a;
f = f / 7.5 * 3;
spartysays “hi “ + e;
gowhite;
```

The language separates instructions with semicolons, just like in C/C++ or Rust. Instead of having a **`main()`** function, the execution starting point in *SpartyTalk* is the **`gogreen`** instruction, and the end of the execution is determined by the **`gowhite`** instruction.

SpartyTalk is a *strongly-typed* language, and it does not like *type inference* like Python or Rust (i.e., guessing types based on the assigned value). Instead, it requires explicit specification of type during the declaration/initialization of a variable, just like in C/C++. We currently have two basic data types: *numbers* and *strings*. We use the **`svar`** keyword to declare a string variable and **`nvar`** keyword for declaring a numeric variable. *SpartyTalk* does not like ambiguity, so the variables must be initialized (e.g., assigned a value during the declaration).

SpartyTalk produces output using the **`spartysays`** command. Also, our language currently supports four operations: **`+`**, **`-`**, **`*`**, and **`/`**. If **`+`** is used with numbers, it performs arithmetic addition. If **`+`** is used with strings, it performs concatenation. If **`+`** is used between a string and a number, it converts the number to a string and performs concatenation (like in Python).

In this project, we will implement a semantic-free trace-only stateless parser for the basic SpartyTalk BNF grammar, as shown below. Please note that through the semester we will be adding more rules to this grammar, so this is not the final SpartyTalk grammar.

```
<program> ::= “gogreen” “;” <statements> “gowhite” “;”
<statements> ::= <statement> | <statements> <statement>

<statement> ::= “spartysays” <expression> “;”
              | “nvar” <identifier> “=” <expression> “;”
              | “svar” <identifier> “=” <expression> “;”
              | <identifier> “=” <expression> “;”

<expression> ::= <identifier>
              | <number>
              | <string>
              | “(“ <expression> “)”
              | <expression> “+” <expression>
              | <expression> “-” <expression>
              | <expression> “*” <expression>
              | <expression> “/” <expression>

<identifier> ::= ([a-zA-Z][a-zA-Z0-9]*)
<number> ::= ([+\-]?[0-9]+(\.[0-9]+)?)
<string> ::= ("[^"]*")
```

The goal of this project is to implement the context-free parser that traces the syntactic analysis of the program following the above grammar.

Implementation

Implement the **`parse_spartytalk()`** function in **`solution.py`**. The function takes a single argument — a *SpartyTalk* program in the form of a string. The function does not return anything. Instead, it prints the trace of parsing or throws an exception (e.g., a **ValueError**) if parsing fails.

Each grammar rule, as being executed during parsing, produces one line of trace printed to the standard output. The following is the format of the trace for each production:

```
<rule_name> ::= list_of_right_side_items_separated_with_spaces
```

For example:

```
<expression> ::= Token('OPEN_PARENS', '(') <expression> Token('CLOSE_PARENS', ')')
```

If the item on the right-hand side is a production rule, you should enclose its name in angle brackets. If the production is a lexical token, it should be presented in the **`Token('LEXEME', 'VALUE')`** format. Please see the open tests to better understand the requirements of the tracing format.

Testing and Grading

The solution will be graded using 40 autograding tests: 20 tests in **`test_open.py`** and 20 additional hidden tests that will be used by the instructors while grading. The hidden tests *will not* introduce any new challenges on top of the ones already tested by the open tests. To run the tests, run the **`pytest`** command while in the project directory. Each test is worth 2 points, resulting in 80 total possible points. Please read the open tests to better understand the requirements of the implementation, *but do not modify the tests*. **Submit the solution to D2L.**

Have fun!