Project 1: Implementing Lexer for SpartyTalk

DUE: Sep 16, 2022 11:59 PM EDT**. *Please make sure to always make backups of your files!***

Description

In this project, we begin implementing the language called *SpartyTalk* by scanning its source code for tokens corresponding to 16 basic lexemes. The lexemes we are going to use in this project is only a subset of the full set of lexemes that will be introduced throughout the semester. The basic 16 lexemes allow to write simple programs like the one below:

```
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).

However, all the above rules are just for your future reference, and they do not play any role in this project. The goal of this project is to create a *SpartyTalk lexer* and use this lexer to extract *tokens* from source code. Our tokens will be based on the 16 basic lexemes described in the table below.

Lexeme Name	Meaning	Format
`GOGREEN`	Start the program	Lowercase **`gogreen`** command
`GOWHITE`	End of the program	Lowercase **`gowhite`** command
`SPARTYSAYS`	Output a string and add a newline	Lowercase **`spartysays`** command
`SEMICOLON`	End of the current instruction	Just a **`;`** sign, like in C or Rust
`NVAR`	Declare a numerical variable	Lowercase **`nvar`** command
`SVAR`	Declare a string variable	Lowercase **`svar`** command
`IDENTIFIER`	Identifiers are used for variable names; they will also be used for function names in the future	Case-sensitive. Must start with a lowercase or uppercase Latin/English letter. It can use digits, but it cannot start with a digit. No other symbols are allowed except letters and digits.
`NUMBER`	A literal of a unified numeric type (used for both integers, floating point numbers, and big numbers)	Can be an integer (e.g., **`10`**), or a floating point number (e.g., **`3.14`**). Can be negative (e.g., **`-3.14`**) or explicitly positive (e.g., **`+3.14`**). If no **`+`** or **`-`** signs are provided, the number is assumed to be non-negative. No scientific notation or other symbols (except for digits, **`.`** (dot), **`+`**, or **`-`**) are allowed. The number cannot start with a dot (i.e., use **`0.5`**, *not* **`.5`**). The number must not end with a dot either (i.e., use **`10`** instead of **`10.`**)
`STRING`	A double-quote style string literal.	Lexically, everything between two double quotes is a string literal. We will deal with escape characters later in the semester.
`PLUS`	Plus operation is used for arithmetic addition and string concatenation	Just a **`+`** sign. Not to be confused with the explicit positive sign as part of the **NUMBER** lexeme.
`MINUS`	Minus arithmetic operation	Just a **`-`** sign.Not to be confused with the negative sign as part of the **NUMBER** lexeme.
`MUL`	Arithmetic multiplication	Just a **`*`** sign.
`DIV`	Arithmetic division	Just a **`/`** sign.
`ASSIGNMENT`	Assigns the value of the right expression to the variable on the left	Just a **`=`** sign.
`OPEN_PARENS`	Opening parenthesis	Just a **`(`** sign.
`CLOSE_PARENS`	Closing parenthesis	Just a **`)`** sign.

Implementation

Implement the **lex_spartytalk()** function in **solution.py**. The function takes a single argument — a *SpartyTalk* program in the form of a string. The function returns a 3-tuple with the following fields:

- **Field 1**: List of objects of the **Token** class if lexing succeeds, and **None** if lexing fails due to the **LexingError** exception.
- **Field 2**: This field has **-1** if lexing succeeds. If lexing fails, it has the line number at which lexing failed.
- **Field 3**: This field has **-1** if lexing succeeds. If lexing fails, it has the column number at which lexing failed.

Note: **lex_spartytalk()** should catch **LexingError** exceptions, and it must not raise any exceptions by itself.

This is an example of an output tuple of **lex_spartytalk()** in case of success:

```
(
    [   Token('GOGREEN', 'gogreen'),
        Token('SEMICOLON', ';'),
        Token('GOWHITE', 'gowhite'),
        Token('SEMICOLON', ';')
    ],
    -1,
    -1
)
```

And this is an example of an output tuple of **lex_spartytalk()** in case of lexing error:

(None, 3, 2)

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!