Project 7: Implementing Loops and Functions in SpartyTalk

DUE: December 9, 2022 11:59 PM EST**.** ***Please make sure to always have backup copies of your files! The last possible date for the late submission of this project (or any other assignment) is December 12th (11:59 PM EST).***

Description

In this project, we will finish implementing the language called *SpartyTalk* by adding loops and functions to our language. The implementation of this project should be based on the lexer implemented in Project 1, parser implemented in Project 3, interpreter implemented in Project 5, and boolean expressions and scopes implemented in Project 6. Also, in this project you will further modify the grammar of the language to enable the new features.

First, let us take a look at the improved *SpartyTalk*, which now has loops and functions. 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;
    nvar e = a * 2;
    nvar f = 3.5 / a;
    f = f / 7.5 * 3;
    spartysays "hi " + e;
    if a < f gogreen;
        spartysays "a is less than f";
    gowhite; else gogreen;
        spartysays "a is greater or equal than f";
    gowhite;


    function foo (a, b) gogreen;
        nvar c = a;
        nvar d = b;
        nvar e = a + b;
        spartysays "a=" + a;
        spartysays "b=" + b;
```

```
        spartysays "e=" + e;
        return e;
    gowhite;

    function bar(a) gogreen;
        return "hello";
    gowhite;

    function baz() gogreen;
        return 7;
    gowhite;

    nvar i = 0;

    while i < 10 gogreen;
        spartysays i;
        i = i + 1;
        if i < 100 gogreen;
            spartysays call bar("aaa");
        gowhite;
    gowhite;

    svar ss = call foo(3, 10);
    spartysays ss;

    spartysays call baz();
gowhite;
```

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

SpartyTalk is a *strongly-typed* language, and it does not do *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 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. The variables must be initialized (e.g., assigned a value during the declaration). Also, our language is quite strict about type conversion. To convert a value from a number to a string, we use the **`svar s = n;`** syntax, where **`n`** is a numeric variable (not literal or expression). To convert a value from a string to a number, use the **`nvar n = s;`** syntax, where **`s`** is a string variable (not expression).

SpartyTalk produces output using the **`spartysays`** command. The argument of this command must be a string expression, a single number, single numeric variable, or an expression that evaluates to a string.

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). Please check the tests carefully to understand the expected behavior of *SpartyTalk* expressions.

SpartyTalk supports branching using **`if`** and **`if...else`** statements. An **`if`** keyword is immediately followed by a boolean expression, and this boolean expression is followed by a scope. Similarly, in an **`if...else`** statement, **`if`** is followed by a boolean expression, after which goes a scope (the true scope) followed by an **`else`** statement, after which goes another scope (the false scope). Please see the above example and the assignment tests to make sense of how the branching is expected to work.

SpartyTalk now supports loops using the **`while`** statement. A **`while`** keyword is immediately followed by a boolean expression, and this boolean expression is followed by a scope. The scope is the body of the loop that executes as long as the boolean expression evaluates to true.

SpartyTalk also now supports functions using the **`function`** statement (for declaring a function) and the **`call`** statement (for calling a function). Also, in this project we introduce the **`return`** statement for returning a value of an evaluated expression from the function. The **`function`** statement is followed by a function name (identifier) and the list of zero or more comma-separated parameters inside parentheses, after which goes the function's scope. Each parameter is a variable name (identifier). The current version of *SpartyTalk* assumes that every parameter has the string type. If you want to perform a mathematical computation with an argument of a function, you need to convert the string value to a number first. The **`call`** keyword can be both a part of a statement or a part of an expression. In both cases, it is followed by a list of zero or more arguments inside a pair of parentheses. Each argument is an expression. Please refer to the example above and the test cases to learn more about the definition and invocation of functions in *SpartyTalk*.

In this project, you are expected to further modify the grammar of the language to incorporate loops and functions. Like in the previous project, you will do the modification of the grammar by yourself. You are expected to continue implementing the function **`interpret_spartytalk().`** The function has the same input-output specifications as in Project 5 and Project 6. Please note that **`interpret_spartytalk()`** is not a scope interpretation function.

For example, if the argument of **`interpret_spartytalk()`** is this program:

```
gogreen;
    nvar sum = 0;
    nvar i = 1;
    while i <= 100 gogreen;
        sum = sum + i;
        i = i + 1;
    gowhite;
    spartysays sum;
gowhite;
```

Then the result of the execution should be the following string printed to the standard output (followed by a newline, like in Python's **`print()`**):

```
5050
```

Please examine the assignment's tests to learn how scope binding and shadowing works in *SpartyTalk*.

Implementation

Continue the implementation of the **`interpret_spartytalk()`** function in **`solution.py`**. Implement the keywords **`while`**, **`function`**, **`call`**, and **`return`**, and make them interpretable per the above specifications.

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, invoke 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!