Project 4: Stateful Parsing and Runtime Environment for SpartyTalk

DUE: October 28, 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 adding a state to the parser and preparing the runtime environment for interpretation. The implementation of this project should be based on the lexer implemented in Project 1 and the parser implemented in Project 3. Essentially, this project adds new information to the intermediate representation and creates a new function for interpreting a program in the intermediate representation form.

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 + a);
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 starting point of execution 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 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 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 continue implementing the parser for the basic *SpartyTalk* BNF grammar, as shown below. Please note that throughout 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> ::= ("[^"]*")
```

This program has two goals:

1. Add the new item, called **“id”**, at the beginning of every JSON sub-object representing a statement or expression in the IR. Each ID is a unique integer representing the order that the statement or expression is processed by the parser, which is essentially the depth-first left-to-right post-order traversal of the AST.
2. Create a new function, called **`interpret_spartytalk()`**, which returns the list of statement IDs in the exact order of their *execution* (not parsing!). This function has one argument — the IR of the program in JSON format.

For instance, the new intermediate representation corresponding to the example program at the beginning of this document is as follows:

```
{
    "type": "program",
    "statements": [
        {
            "id": 2,
            "type": "statement",
            "statement_type": "nvar",
            "identifier": "a",
            "expression": {
                "id": 1,
                "type": "expression",
                "expression_type": "number",
                "value": "-10.5"
            }
        },
        {
            "id": 4,
            "type": "statement",
            "statement_type": "svar",
            "identifier": "b",
            "expression": {
                "id": 3,
                "type": "expression",
                "expression_type": "string",
                "value": "hello\n"
            }
        },
        {
            "id": 6,
            "type": "statement",
            "statement_type": "svar",
            "identifier": "c",
            "expression": {
                "id": 5,
                "type": "expression",
                "expression_type": "string",
                "value": "world"
            }
        },
        {
            "id": 13,
            "type": "statement",
            "statement_type": "svar",
            "identifier": "d",
            "expression": {
                "id": 12,
                "type": "expression",
                "expression_type": "plus",
                "left": {
                    "id": 7,
                    "type": "expression",
                    "expression_type": "identifier",
                    "identifier": "b"
                },
                "right": {
                    "id": 11,
                    "type": "expression",
                    "expression_type": "parentheses",
                    "expression": {
                        "id": 10,
                        "type": "expression",
                        "expression_type": "plus",
                        "left": {
                            "id": 8,
                            "type": "expression",
                            "expression_type": "identifier",
                            "identifier": "c"
                        },
                        "right": {
                            "id": 9,
                            "type": "expression",
                            "expression_type": "identifier",
                            "identifier": "a"
                        }
```

```
                    }
                }
            }
        },
        {
            "id": 17,
            "type": "statement",
            "statement_type": "nvar",
            "identifier": "e",
            "expression": {
                "id": 16,
                "type": "expression",
                "expression_type": "mul",
                "left": {
                    "id": 14,
                    "type": "expression",
                    "expression_type": "identifier",
                    "identifier": "a"
                },
                "right": {
                    "id": 15,
                    "type": "expression",
                    "expression_type": "number",
                    "value": "2"
                }
            }
        },
        {
            "id": 21,
            "type": "statement",
            "statement_type": "nvar",
            "identifier": "f",
            "expression": {
                "id": 20,
                "type": "expression",
                "expression_type": "div",
                "left": {
                    "id": 18,
                    "type": "expression",
                    "expression_type": "number",
                    "value": "3.5"
                },
                "right": {
                    "id": 19,
                    "type": "expression",
                    "expression_type": "identifier",
                    "identifier": "a"
                }
            }
        },
        {
            "id": 27,
            "type": "statement",
            "statement_type": "assignment",
            "identifier": "f",
            "expression": {
                "id": 26,
                "type": "expression",
                "expression_type": "mul",
                "left": {
                    "id": 24,
                    "type": "expression",
                    "expression_type": "div",
                    "left": {
                        "id": 22,
                        "type": "expression",
                        "expression_type": "identifier",
                        "identifier": "f"
                    },
                    "right": {
                        "id": 23,
                        "type": "expression",
                        "expression_type": "number",
                        "value": "7.5"
                    }
                },
```

```
                "right": {
                    "id": 25,
                    "type": "expression",
                    "expression_type": "number",
                    "value": "3"
                }
            }
        },
        {
            "id": 31,
            "type": "statement",
            "statement_type": "spartysays",
            "expression": {
                "id": 30,
                "type": "expression",
                "expression_type": "plus",
                "left": {
                    "id": 28,
                    "type": "expression",
                    "expression_type": "string",
                    "value": "hi "
                },
                "right": {
                    "id": 29,
                    "type": "expression",
                    "expression_type": "identifier",
                    "identifier": "e"
                }
            }
        }
    ]
}
```

The sequence of statement IDs, returned by the **`interpret_spartytalk()`** function would be as follows:

`[2, 4, 6, 13, 17, 21, 27, 31]`

Implementation

Continue the implementation of **`parse_spartytalk()`** function in **`solution.py`**. The function takes a single argument — a *SpartyTalk* program in the form of a string. In Project 2, this function did not return anything. However, in this project, **`parse_spartytalk()`** should return the JSON object with the intermediate representation (IR) of the program if parsing succeeds. In this project, we do not expect **`parse_spartytalk()`** to print anything to the standard output. If parsing fails, the function raises an **`Exception`** with one argument, which is a JSON object following the format below:

```
{
    “type”: “error”,
    “tokentype”: <token_type>,
    “line”: <line_number>,
    “column”: <column_number>,
    “id”: <element_id>
}
```

Where **<line_number>** and **<column_number>** are the line and column, respectively, where the error occurred. **<token_type>** is the output of **gettokentype()** function of the **Token** class object associated with the error. **<element_id>** is the ID of the statement or expression associated with or directly preceding the error (or **0** if there is no associated or preceding statement/expression).

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!