Project 3: Creating an Intermediate Representation of a SpartyTalk program

DUE: October 14, 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 *creating an intermediate representation of its source code.* The implementation of this project should be based on the lexer implemented in Project 1 and the parser-tracer implemented in Project 2. Essentially, this project adds new features to the parser.

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> ::= ("[^"]*")
```

The goal of this project is to make the parser produce an intermediate representation (IR) of the program in JSON format following the specifications outlined in the table below.

IR Item	JSON Type	Format	Description

Program object	object	`{`
		`    “type”: “program”,`
		`    “statements”: <list_of_statements>`
		`}`
			This is the root (parent) level of the intermediate representation. **`<list_of_statements>`** is a JSON array of JSON objects representing the statements of the program.

Main statements of the program	array	`[`
		`    <statement>,`
		`    <statement>,`
		`    . . .`
		`]`
			This array has all the statements between **`gogreen`** and **`gowhite`**, but not including **`gogreen`** and **`gowhite`**. Each **`<statement>`** is a JSON object representing a statement.

`spartysays` statement	object	`{`
		`    “type”: “statement”,`
		`    “statement_type”: “spartysays”,`
		`    “expression”: <expression>`
		`}`
			This object represents the spartysays statement. The **`<expression>`** is a JSON object representing the argument of **`spartysays`**.

`nvar` statement	object	`{`
		`    “type”: “statement”,`
		`    “statement_type”: “nvar”,`
		`    “identifier”: <identifier>`
		`    “expression”: <expression>`
		`}`
			This object represents the **`nvar`** statement. **`<identifier>`** is a JSON string representing the name of the variable. **`<expression>`** is a JSON object representing the initialization value of the **`nvar`** statement.

`svar` statement	object	`{`
		`    “type”: “statement”,`
		`    “statement_type”: “svar”,`
		`    “identifier”: <identifier>`
		`    “expression”: <expression>`
		`}`
			This object represents the **`svar`** statement. **`<identifier>`** is a JSON string representing the name of the variable. **`<expression>`** is a JSON object representing the initialization value of the **`svar`** statement.

Variable assignment statement	object	`{`
		`    “type”: “statement”,`
		`    “statement_type”: “assignment”,`
		`    “identifier”: <identifier>`
		`    “expression”: <expression>`
		`}`
			This object represents a variable assignment statement. **`<identifier>`** is a JSON string representing the name of the variable. **`<expression>`** is a JSON object representing the assigned value.

Terminal identifier expression	object	`{`
		`    “type”: “expression”,`
		`    “expression_type”: “identifier”,`
		`    “identifier”: <identifier>`
		`}`
			This object represents the expression consisting of a single identifier (i.e., variable name). **`<identifier>`** is a JSON string representing the identifier name.

Terminal numeric expression	object	`{`
		    `“type”: “expression”,`
		    `“expression_type”: “number”,`
		    `“value”: <value>`
		`}`
			This object represents the expression consisting of a single number. **`<value>`** is the string representing the value of the number. Although JSON supports integer and floating point numbers, we still use string for the numeric value to guarantee safe passing of unconventional numeric formats.

Terminal string expression	object	`{`
		    `“type”: “expression”,`
		    `“expression_type”: “string”,`
		    `“value”: <value>`
		`}`
			This object represents the expression consisting of a single string literal. **`<value>`** is the string representing the value of the string.

Expression in parentheses	object	`{`
		    `“type”: “expression”,`
		    `“expression_type”: “parentheses”,`
		    `“expression”: <expression>`
		`}`
			This object represents the expression **`<expression>`** enclosed in a pair of parentheses. **`<expression>`** is a JSON object.

Plus expression	object	`{`
		    `“type”: “expression”,`
		    `“expression_type”: “plus”,`
		    `“left”: <left_expression>,`
		    `“right”: <right_expression>`
		`}`
			This object represents the plus operation between **`<left_expression>`** and **`<right_expression>`**, which are both JSON objects.

Minus expression	object	`{`
		    `“type”: “expression”,`
		    `“expression_type”: “minus”,`
		    `“left”: <left_expression>,`
		    `“right”: <right_expression>`
		`}`
			This object represents the minus operation between **`<left_expression>`** and **`<right_expression>`**, which are both JSON objects.

Multiplication expression	object	`{`
		    `“type”: “expression”,`
		    `“expression_type”: “mul”,`
		    `“left”: <left_expression>,`
		    `“right”: <right_expression>`
		`}`
			This object represents the multiplication operation between **`<left_expression>`** and **`<right_expression>`**, which are both JSON objects.

Division expression	object	`{`
		    `“type”: “expression”,`
		    `“expression_type”: “div”,`
		    `“left”: <left_expression>,`
		    `“right”: <right_expression>`
		`}`
			This object represents the division operation between **`<left_expression>`** and **`<right_expression>`**, which are both JSON objects.

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

```
{
    "type": "program",
    "statements": [
        {
            "type": "statement",
            "statement_type": "nvar",
            "identifier": "a",
            "expression": {
                "type": "expression",
                "expression_type": "number",
                "value": "-10.5"
            }
        },
        {
            "type": "statement",
            "statement_type": "svar",
            "identifier": "b",
            "expression": {
                "type": "expression",
                "expression_type": "string",
                "value": "hello\n"
            }
        },
        {
            "type": "statement",
            "statement_type": "svar",
            "identifier": "c",
            "expression": {
                "type": "expression",
                "expression_type": "string",
                "value": "world"
            }
        },
        {
            "type": "statement",
            "statement_type": "svar",
            "identifier": "d",
            "expression": {
                "type": "expression",
                "expression_type": "plus",
                "left": {
                    "type": "expression",
                    "expression_type": "identifier",
                    "identifier": "b"
                },
                "right": {
                    "type": "expression",
                    "expression_type": "parentheses",
                    "expression": {
                        "type": "expression",
                        "expression_type": "plus",
                        "left": {
                            "type": "expression",
                            "expression_type": "identifier",
                            "identifier": "c"
                        },
                        "right": {
                            "type": "expression",
                            "expression_type": "identifier",
                            "identifier": "a"
                        }
                    }
                }
            }
        },
        {
            "type": "statement",
            "statement_type": "nvar",
            "identifier": "e",
            "expression": {
                "type": "expression",
                "expression_type": "mul",
                "left": {
                    "type": "expression",
                    "expression_type": "identifier",
                    "identifier": "a"
                },
                "right": {
                    "type": "expression",
                    "expression_type": "number",
                    "value": "2"
                }
            }
        },
        {
            "type": "statement",
            "statement_type": "nvar",
            "identifier": "f",
            "expression": {
                "type": "expression",
                "expression_type": "div",
                "left": {
                    "type": "expression",
                    "expression_type": "number",
                    "value": "3.5"
                },
                "right": {
                    "type": "expression",
                    "expression_type": "identifier",
                    "identifier": "a"
                }
            }
        },
        {
            "type": "statement",
            "statement_type": "assignment",
            "identifier": "f",
            "expression": {
                "type": "expression",
                "expression_type": "mul",
                "left": {
                    "type": "expression",
                    "expression_type": "div",
                    "left": {
                        "type": "expression",
                        "expression_type": "identifier",
                        "identifier": "f"
                    },
                    "right": {
                        "type": "expression",
                        "expression_type": "number",
                        "value": "7.5"
                    }
                },
                "right": {
                    "type": "expression",
                    "expression_type": "number",
                    "value": "3"
                }
            }
        },
        {
            "type": "statement",
            "statement_type": "spartysays",
            "expression": {
                "type": "expression",
                "expression_type": "plus",
                "left": {
                    "type": "expression",
                    "expression_type": "string",
                    "value": "hi "
                },
                "right": {
                    "type": "expression",
                    "expression_type": "identifier",
                    "identifier": "e"
                }
            }
        }
    ]
}
```

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>
}
```

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.

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!