Assignment 3b: A Simple Programming Language

Data Structures, CSCI-UA 102, Section 7
Instructor: Max Sklar

Use the arithmetic expression interpreter from assignment 3a to build a simple interpreted programming environment.

Goals

- Understand the process of evaluating expressions and binding variables.
- Implementing the Map (Key-Value lookup) Interface using either a binary search tree or a hash table.

Overview

A *compiled language* is one in which source code is converted into machine code before execution. Java is a compiled language, and the `javac` command is used to compile your Java source code into byte code stored in .class files.

The alternative is an *interpreted language* where the source code is analyzed and executed directly by its *interpreter*. One example is Javascript, which is downloaded and interpreted by web browsers on the fly. Python is also interpreted.

In assignment 3a, you built an expression interpreter that turned an arithmetic expression into an expression tree. You then evaluated that tree. If variables were present, you identified those as *unbound variables* and did not complete the evaluation. An *unbound variable* is a variable that does not have a value. In Java, an attempt to access an unbound variable in an expression will not compile.

A *bound variable* is one that has a value assigned, which is typically stored somewhere in memory. You can imagine that under the hood, the interpreter maintains a mapping of all of the bound variables and their values. When these variables appear in the code, their appropriate value can be looked up and used in real time.

In this project, you will expand your expression interpreter to include a REPL, so that you can set variables and plug them into expressions. Your language won't be Turing-complete in that it won't be able to handle loops or recursion. But with the simple changes specified in this document, it will be able to handle far more complex mathematical formulas.

The Repl

The REPL, or **Read-Evaluate-Print Loop**, is an interactive environment that reads user input, evaluates it, and then prints the result. This type of loop is common in many interpreted languages, allowing users to execute commands or expressions one at a time and immediately view the output.

In this assignment, a basic REPL is provided in the main function of ExpressionRepl.java. The main function initializes the REPL and maintains an instance object, repl. The method `evaluateString` on `repl` takes a command (the input) and returns a String representing the output, allowing users to evaluate expressions and see results in real time.

The New Tokenizer

There is a new version of the tokenizer for this assignment. Instead of returning a `SinglyLinkedList<Strin` it returns an internal class `TokenList`, which contains two fields:

```
final public String variableName;
final public SinglyLinkedList<String> list;
```

The first field `list` contains the list of tokens that the original tokenizer returned. The second field, `variableName`, stores the name of a variable if one is being assigned.

The tokenizer now accepts input in one of two forms:

1. `<variableName> = <expression>` — in this case, `variableName` will store the variable name, and `list` will store the tokens for `<expression>`.
2. `<expression>` — here, only `list` will be populated with tokens representing the expression, and `variableName` will be left as `null`

For example, consider the following input:

```
a = (2 + 3) ^ 3 - 1.
```

The tokenizer will produce a `TokenList` where `variableName` is `"a"`, and `list` contains the tokens for

```
(2 + 3) ^ 3 - 1.
```

This structure allows the REPL to distinguish between expressions that assign a value to a variable and those that are simply evaluated without assignment.

Your Tasks

The following needs to be done in order to start this assignment:

- Start with a copy of your working code from assignment 3a and import the given files ExpressionRepl.java, MapInterface.java, MapImplemented.java, and Tokenizer.java.

- Because the tokenizer has changed, you will get a compile error in ExpressionInterpreter.java on the line:

  ```
  SinglyLinkedList<String> tokens = Tokenizer.tokenize(args[0]);
  ```

  To fix this, add `.list` to the tokenized result as follows in order to get the list of tokens from an instance of `TokenList`, which is what `tokenize` now returns.

  ```
  SinglyLinkedList<String> tokens = Tokenizer.tokenize(args[0]).list;
  ```

Next, read through ExpressionRepl.java. The function `evaulateInput` builds an expression tree, evaluates that tree, and finally if there's a variable involved, it sets the variable to that result. In order to get it working, two parts need to be completed.

1. The field `variableBindings` is a Map from Strings (variables) to Doubles. It needs to be implemented! This happens in MapImplemented.java, where the put function and the get function have been left blank. There are several ways to implement a map. Full credit will be given for any implemention of a Binary Search Tree or a Hash table. You may use textbook code for reference. Implementations using an inefficient data structure such as a list or array will work and be given partial credit.

2. Coming back to ExpressionRepl.java, `solveIfPossible` needs to work in solving expressions. In ExpressionInterpreter we have `solveAsMuchAsPossible` which simply gave up if confronted with a variable. Fortunately, in `solveIfPossible`, we can use our variable bindings (through the `get` function) to find out if a variable has previously been set. If so, we can use those values. If not, we can print that we encountered an unbound variable and return null.

The Test Class

You are also given ExpressionReplTest.java, which you can use to test your code. You can use it by simply compiling and running the test class:

```
javac ExpressionReplTest.java
java ExpressionReplTest
```

This will kick off 6 tests, and if you have implemented everything properly they should all pass. The final two tests try to bind a large number of variables which tests the limit of your map implementation. The test also prints the timing of these operations so that you can see if your map implementation is efficient. If it is, then the final test should finish in a matter of seconds.

Feel free to include additional tests.

Sample Repl Session

```
java ExpressionRepl
> 5 + 6 * 2
17.0
> x = (35 - 8)^2
x = 729.0
> y = x / z
Unbound Variable: z
Not solved.
> y
Unbound Variable: y
Not solved.
> y = x / 3
y = 243.0
> z = 10
z = 10.0
> z = z + 1
z = 11.0
> x
729.0
> y
243.0
> z
11.0
```

README

Included in the `readme.txt` file should be the following:

RUNBOOK: Information on how to compile and run your code in the command line.

TIME SPENT: An estimate of the time spent working on this project.

NOTES: Include the difficulties and roadblocks encountered while working on this project, including notable bugs that needed to be overcome. Are there any quirks in your solution that we should be aware of? Was the outcome what you expected?

RESOURCES AND ACKNOWLEDGEMENTS: Disclose the sources you consulted for help on this project. Possible human sources include: NYU Staff (the instructor, TA, tutor), other students, and anyone who helped you with the project. Sources that you consulted, including code that you found online, in the book, and output from LLMs. Describe the nature of your LLM usage.

There will never be a grading penalty for a complete and accurate accounting of resources and acknowledgments.