San Francisco State University
CSC 413
Dawson Zhou (zhoud@sfsu.edu)

Assignment 2: Interpreter

Overview

The goal of this assignment is to apply design principles we've learned in class to develop a complex Interpreter program while maintaining readability. The assignment is heavily based on the Interpreter demo that we worked on during class in Lecture 11 and Lecture 12, with additional features described in this handout and in Lecture 16.

Submission

Assignment 2 is due on **Monday, November 2, 2020** at **11:59 pm PT**.
Late submissions will be accepted with a penalty:

- November 3, 2020 12:00 am PT to November 4, 2020 11:59 pm PT will receive 80%
- November 5, 2020 12:00 am PT to November 6, 2020 11:59 pm PT will receive 50%

Grading

100 pts total:

- 80 pts: Implementation of features
    - 10 pts: **ProgramState** implemented correctly
        - 4 pts: Handles setting and reading variables properly
        - 4 pts: Handles setting and reading function parameter names and definitions properly
        - 2 pts: Supports a stack of frames for scoping variables to function calls
    - 40 pts: All **Statement** subclasses implemented correctly, including:
        - 2 pts: **PrintStatement**
        - 2 pts: **AssignStatement**
        - 4 pts: **IfStatement**
        - 4 pts: **WhileStatement**
        - 7 pts: **ForStatement**
        - 7 pts: **DefineFunctionStatement**
        - 2 pts: **ReturnStatement**

        - 4 pts: Use of a **BlockStatement** abstract class for **IfStatement**, **WhileStatement**, and **ForStatement** (see IfStatement, WhileStatement, ForStatement for details)
        - 4 pts: Use of a **LoopStatement** abstract class for **WhileStatement** and **ForStatement** (see IfStatement, WhileStatement, ForStatement for details)
        - 4 pts: Properly handling an early **ReturnStatement** in each of **IfStatement**, **WhileStatement**, and **ForStatement**
    - 20 pts: All **Expression** subclasses implemented correctly, including:
        - 2 pts: **ConstantExpression**
        - 5 pts: **VariableExpression**
        - 5 pts: **ArithmeticExpression**
        - 8 pts: **FunctionExpression**
    - 5 pts: Missing parts of **Interpreter** implemented correctly
    - 5 pts: Missing parts of **Expression.create** implemented correctly
- 20 pts: Implementation of unit tests for features
    - 4 pts: Unit tests for **ProgramState** class
    - 8 pts: Unit tests for **Statement** subclasses
        - Note: no tests needed for **Statement** itself
    - 8 pts: Unit tests for **Expression** subclasses
        - Note: no tests needed for **Expression** itself

Detailed Requirements

In this assignment, we'll implement an Interpreter class which can parse and run simple programs written in a custom language. The language will have support for:

- Evaluating various expression types, including arithmetic expressions, and variable expressions, to integer values based on the program's state
- Running various statement types, including assign statements and print statements
- Logic branching with if statements, while loops, and for loops
- Defining functions with names, parameter lists, and function bodies, and return statements
- Calling and evaluating functions to integer values
- Scoping function calls with stack frames so that variables defined and modified during a function call do not affect existing program state

To begin with, I strongly recommend reviewing material from Lecture 11 and Lecture 12. We will be using roughly the same class hierarchy structures, but with some modifications to support additional functionality. For this reason, you won't be able to directly copy the demo code from those lectures, but you'll still be able to reference them while you work through the assignment.

I would also strongly recommend reviewing material from Lecture 16, where I introduced and discussed an overview of this assignment.

You will be provided with some full or partially implemented classes: **Interpreter**, **ProgramState**, **Expression**, **ArithmeticExpression**, **Condition**, **Statement**, **BlockStatement**, and **LoopStatement**. Some of these classes will have certain sections marked with a comment:

```
// TODO: Implement.
```

These sections require you to fill in the remaining implementation to complete the class. In other cases, you will need to add and fully implement classes yourself (in particular, other **Statement** and **Expression** subclasses).

I've provided certain interfaces for the various classes that you *should not* modify. The signature of any public or protected method that is not a constructor **cannot be changed**. That includes visibility modifiers, method names, and parameter lists. Changing these will result in you losing points on the assignment, with the exact amount varying based on the extent of the changes.

Along the same lines, any instance variables you declare in any classes for the assignment **must be marked private**. Failure to do so will also result in points being deducted.

Over the next few sections of this document, I'll describe the order in which I would recommend approaching the assignment.

PrintStatement and ConstantExpression ✓

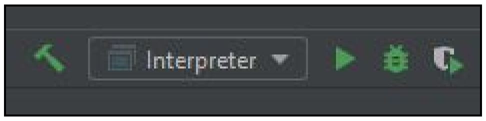
The simplest **Statement** type to implement is the **PrintStatement**, along with the **ConstantExpression** to provide something to print. Note that even if you don't have all **Expression** types implemented, you can still use the **Expression** interface from within a **PrintStatement** to evaluate it as an integer.

In order to be able to run a program using **PrintStatement** and **ConstantExpression**, you'll also need to complete the corresponding portions of **Interpreter.parsePrintStatement** and **Expression.parseConstantExpression**.

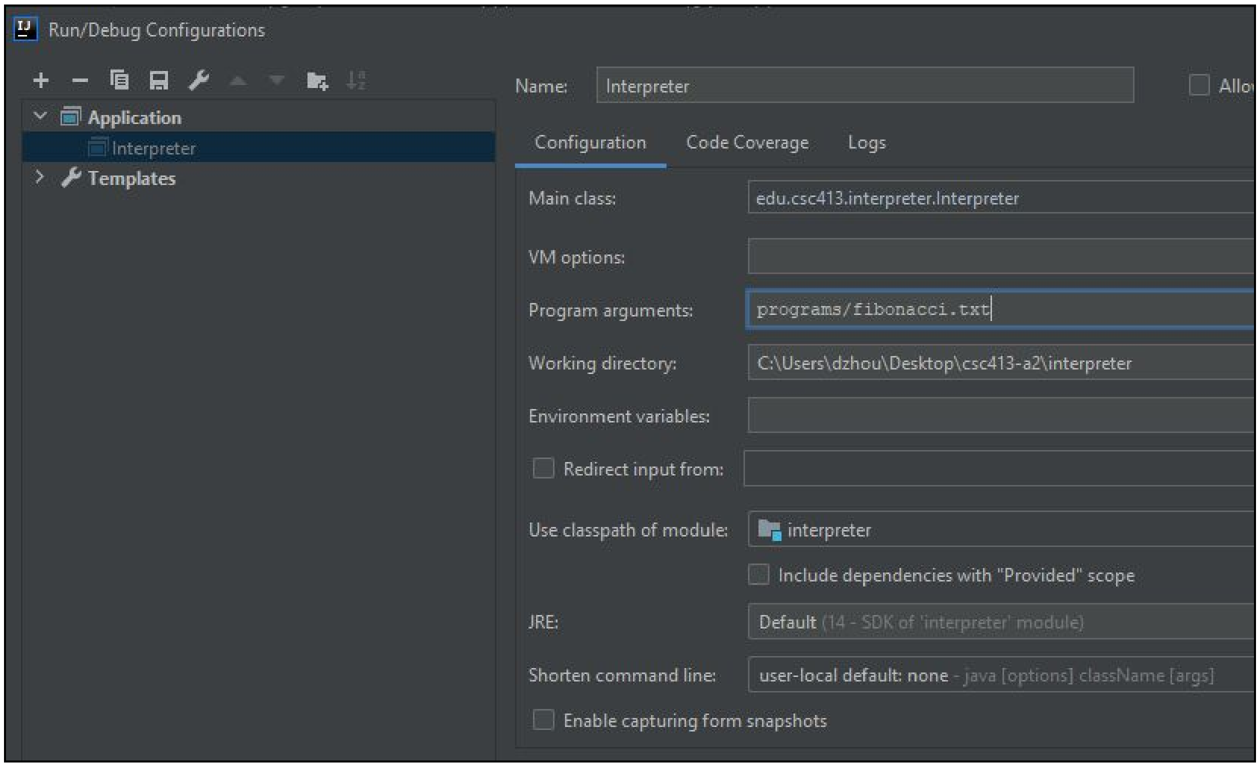
Running a Test Program

I've provided a number of test programs in the "programs/" directory that you can use to test your functionality along the way. Once you've implemented **PrintStatement** and **ConstantExpression**, for example, you can run "programs/simple-print.txt" to test your code.

In order to do so, have a look at the top-right corner of IntelliJ. You should see the following:



If you don't see "Interpreter" in the drop-down menu, you can right-click **Interpreter** in the left-hand project navigation section and select "Run 'Interpreter.main()'". Doing so should result in an error message indicating that you need to provide the file name for a program to run. That's fine -- now you should see "Interpreter" in the drop-down menu in the top-right corner. Click on this menu and select "Edit Configurations" to bring up the following window:



Name: Interpreter

Main class: edu.csc413.interpreter.Interpreter

VM options:

Program arguments: programs/fibonacci.txt

Working directory: C:\Users\dzhou\Desktop\csc413-a2\interpreter

Environment variables:

Redirect input from:

Use classpath of module: interpreter

Include dependencies with "Provided" scope

JRE: Default (14 - SDK of 'interpreter' module)

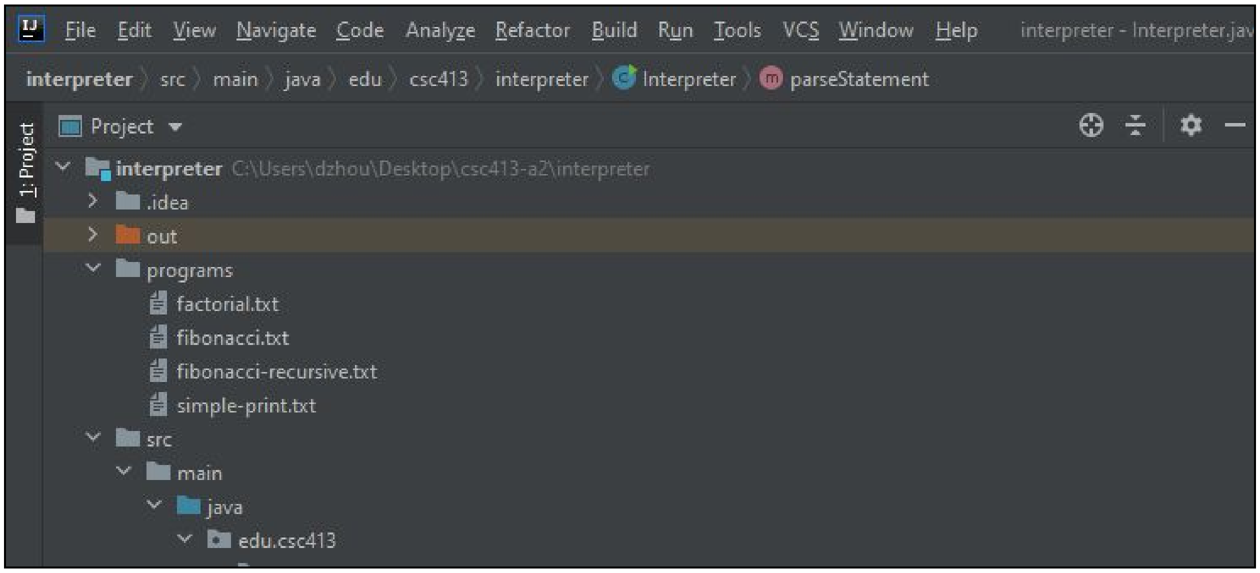
Shorten command line: user-local default: none - java [options] className [args]

Enable capturing form snapshots

Type in the name of the program you want to run in the "Program arguments" field. Remember to include the "programs/" prefix, and make sure that "Working directory" matches what you see above for your project location on your computer (i.e. it should be the .../csc413-a2/interpreter directory).

Select "Okay" to exit the window and hit the green "play" button in the top-right corner next to the drop-down menu to run the program with your newly provided argument. If you want to update which program is run, simply repeat this process; navigate to "Edit Configurations" and update the "Program arguments" file name.

You can see the contents of "programs/simple-print.txt" and other programs by navigating to the "programs" directory in the project navigation section on the left:


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
interpreter - Interpreter.jav
interpreter 〉 src 〉 main 〉 java 〉 edu 〉 csc413 〉 interpreter 〉 Interpreter 〉 parseStatement
1: Project
Project
interpreter C:\Users\dzhou\Desktop\csc413-a2\interpreter
.idea
out
programs
factorial.txt
fibonacci.txt
fibonacci-recursive.txt
simple-print.txt
src
main
java
edu.csc413

Opening one of these files will display the contents in an IntelliJ tab.

AssignStatement, VariableExpression, and ProgramState

✓

The next step to implement is the **AssignStatement**, which relies on **VariableExpression** to be useful. However, in order to accomplish these, we have to consider the **ProgramState** class.

I've provided a skeleton **ProgramState** class with methods defined but no implementation. In order to support **AssignStatement** and **VariableExpression**, you'll need to implement **getVariable** and **setVariable**. You can add whatever instance variables you'll need to track variable names and their corresponding values, but it's important to note that you *must* use the public interface I've provided.

Once you've implemented those portions of ProgramState, you should be able to complete the implementation of **AssignStatement** and **VariableExpression**. As before, you'll need to implement the corresponding parts of **Interpreter.parseAssignStatement** and **Expression.parseVariableExpression** to run a program which uses them.

Once those are done, you should be able to run "programs/assign-print-variables.txt".

ArithmeticExpression ✓

You can implement **ArithmeticExpression** next, which will allow you to assign variables and print expressions that involve arithmetic operators. The operators supported are enumerated in the **ArithmeticExpression.Operator** enum. You'll need to determine the instance variables needed by **ArithmeticExpression** and how its **evaluate** method is implemented (hint: see the demo we went over in lecture).

I've provided most of the logic in parsing an arithmetic expression in **Expression.parseArithmeticExpression**. You'll need to fill in some details to complete it.
Maybe

Once finished, you should be able to run "programs/arithmetic.txt".

IfStatement, WhileStatement, ForStatement ✓

Next, we want to support **IfStatement**, **WhileStatement**, and **ForStatement**. These three statement types have a number of things in common. They all utilize conditional statements, which will be represented in the program by the **Condition** class, which is already implemented for you. They also each contain blocks of **Statements** that may be run.

Rather than implementing them as direct subclasses of **Statement**, you'll need to implement two intermediate abstract classes: **BlockStatement** and **LoopStatement**.

WhileStatement and **ForStatement** are both **LoopStatement**s. The **LoopStatement** class will implement the override of **run**, as both subclasses have the same overall structure -- repeatedly run the block of statements in a loop, checking a condition each iteration to determine if the loop should continue. However, while loops and for loops differ in how they approach **runInitialization** and **runUpdate**. Those are provided as abstract methods in **LoopStatement**. You will be responsible for completing the implementation of **run** in **LoopStatement**, as well as **runInitialization** and **runUpdate** for each of **WhileStatement** and **ForStatement**.

IfStatement and **LoopStatement** are both **BlockStatement**s. The **BlockStatement** class will store the list of **Statement**s comprising a block (the body of the if statement or a loop), and implement a protected method **runBlock** which iterates through those **Statement**s and runs each one.

For each of these classes, please pay close attention to the comments in the provided classes, as they indicate what you are allowed to change and what you need to keep unmodified in order to receive full credit for this portion of the assignment.

Once finished, you should be able to run "programs/loops.txt". ✓

DefineFunctionStatement and ProgramState

Support for defining and calling functions is the largest change from the demo shown in class. It's best to start with being able to parse and run a **DefineFunctionStatement**, which will require modifying **ProgramState**.

Consider the syntax in our custom language for defining a function:

```
define sum(a, b) {
    return a + b
}

print(sum(5, 3))
```

This short program defines the function sum, and then invokes it within a print statement later on. At the time when sum is invoked, the expression (which we will eventually implement as **FunctionExpression**) needs to do the following:

1. Look up the names of the parameters for the function.
2. Match the expressions being passed in as parameters with those names, and initialize them all as variables.
3. Look up the code for the function, in the form of a list of **Statement**s.
4. Invoke **run** on each statement, one after another.
5. When a **ReturnStatement** is encountered, quit running statements in the function and evaluate the original **FunctionExpression** to the value returned by the **ReturnStatement**.

We will address steps 4 and 5 in the next section, but for now, focusing on steps 1, 2, and 3, it's clear that **ProgramState** will need to track some information about the function. When we see the original **DefineFunctionStatement** which provides the function definition, we'll need to remember the function name along with the list of parameter names and the list of **Statement**s comprising the function body.

ProgramState should define instance variables and methods that allow **DefineFunctionStatement** to store that information (function name, parameter names, and function body). In order for us to invoke the function later on, **ProgramState** should define methods that allow program execution to look up the parameter names and the function body given only the name of the function.

Think carefully about how to add instance variables and public methods **ProgramState** to support this functionality.

A Call Stack in ProgramState

The previous section details how we can support defining a function, but we'll still need some additional changes to support actually invoking those functions.

Whenever we invoke a function during program execution, the new function defines its own "scope" where it only has access to parameters and variables defined within that function. For example, consider the following short program:

```
define scope_example(input) {
    x = 5
    return x + input
}

x = 3
print(scope_example(10))
print(x)
```

x is defined as a variable both in the outer scope and in the body of the **scope_example** function. But because the latter is in a separate function, its usage of **x** is completely separate from the former. As a result, the last print statement should print 3 and not the updated 5.

In the Interpreter demo, we used a single map from variable names to values to represent the program's state. That will not be sufficient for this assignment. We'll need to consider the concept of a **call stack**.

In a running program, the call stack is the stack of **call frames** that represent currently executing function calls. Every time we invoke a function, we push a new call frame on top of the call stack to represent that function call's data. All of its parameters and scoped variables are stored in that frame. When the function invocation is completed with a return statement, its corresponding call frame is popped from the top of the call stack and we resume execution of the previous code which originally called the function to begin with.

To implement a call stack for this assignment, we'll use a stack of maps, rather than a single map. Each map is effectively the call frame for a single function invocation. Every time we call a function, we push a new, empty map onto the call stack. Every time a function finishes running, we pop its map from the call stack.

You'll want to start by modifying **ProgramState** to add support for managing a call stack. As described above, you'll want a stack of maps rather than a single map. Reading and writing variable values should happen to the topmost stack, and **ProgramState** should implement the public methods for pushing a new call frame to the stack (**addNewCallFrame**) or popping the topmost call frame from the stack (**removeCurrentCallFrame**).

FunctionExpression

A function call is actually an **Expression**, even though it results in a series of **Statement**s being executed, due to the fact that functions in our custom language all return a value. We define the actual call to the function with the **FunctionExpression** class.

A **FunctionExpression** appears as a function name, followed by a list of parameter **Expression**s in parentheses, separated by commas. When parsed, the **FunctionExpression** will need to store and remember the function name and the parameter **Expression**s.

When evaluated, the **FunctionExpression**, with the methods in **ProgramState** discussed before, should be able to look up the parameter names and **Statement**s associated with the function. It should push a new call frame onto the **ProgramState**'s call stack, assign the parameter **Expression**s to their corresponding names in the new call frame, and then run each statement of the function one at a time.

Note that one of these **Statement**s might be a **ReturnStatement**. That will determine the value to which the **FunctionExpression** eventually evaluates.

ReturnStatement

Within the body of a function, we expect there to be at least one **ReturnStatement** which returns an expression. The **ReturnStatement** being run should result in two things happening:

1. The **ReturnStatement** should evaluate its return expression and indicate in **ProgramState** that that particular value is being returned.

2. The original **FunctionExpression** that executed the **ReturnStatement** should detect in **ProgramState** that a value was returned; it can then stop its own evaluation and return that returned value as the result of **FunctionExpression.evaluate**.

In order to support this, we'll need to implement another group of methods to **ProgramState**: **hasReturnValue**, **getReturnValue**, **setReturnValue**, and **clearReturnValue**.

- The **ReturnStatement**, when run, can call **ProgramState.setReturnValue** to set the integer being returned.
- The **FunctionExpression**, after running each **Statement**, can check **ProgramState.hasReturnValue** to see if a value has been returned.
- If **FunctionExpression** detects that a value has been returned, it needs to cease execution and return that value as the result of its **evaluate** method.
    - It retrieves that return value with **ProgramState.getReturnValue**
    - It calls **ProgramState.clearReturnValue**, as the previous call frame should continue executing afterwards
    - It pops the current call frame from the call stack with **ProgramState.removeCallFrame**
    - It returns the previously retrieved return value

Once all of these are done, you'll finally be able to test functions in a program. At this point, you should be able to run "programs/simple-function.txt".

Revisit IfStatement, WhileStatement, and ForStatement

With the implementation of functions and return statements, we have to consider that **IfStatement**, **WhileStatement**, and **ForStatement** are no longer correct. If a **ReturnStatement** appears in one of their blocks, we expect that the entire block should be finished without running any additional **Statement**s. Further, for the **LoopStatement**s, the loop should not continue, regardless of how the condition evaluates.

Consider the changes you need to make this work. If you arranged things correctly before with the **BlockStatement** and **LoopStatement** abstract classes, this should not involve too many changes!

Unit Tests

Unit tests are not as large of a portion of your grade for this assignment, simply due to the increased complexity of the implementation. However, you'll still be graded on the thoroughness and quality of your unit tests. I strongly encourage you to write tests as you work through the implementation, as it will help you debug any issues that come up during implementation.

You are responsible for writing unit tests for:

- The **ProgramState** class
- All subclasses of **Statement** (but not **Statement** itself)
- All subclasses of **Expression** (but not **Expression** itself)

Your unit tests should be:

- Thorough: cover a range of potential cases (both valid and invalid) to ensure that the class/method behave as expected
- Concise: tests should minimize extraneous setup and input that doesn't relate to what the test focuses on
- Consistent: tests should be named and structured with consistency, which helps a reader understand at a glance what the test is accomplishing
- Correct: the test code should match what the test claims it does

Assignment Setup

Please refer to the handout for Assignment 0 on detailed steps for setting up a private GitHub repository, for setting up an IntelliJ project for the starter code provided on iLearn, and for steps on how to submit your assignment to GitHub.

You are expected to invite the instructor (Git name: dawsonmz) as a collaborator for the private repository you create. Your assignment will be graded based on what is submitted to GitHub.