Phase 4 – SPLAT Executor

Target Date: Friday, November 24

PHASE 4 SETUP

Starting with the same Java project from the previous task, add in the provided package folder, **splat.executor** found in the archive file executor_start.zip. As always, make sure to preserve the proper package structure in **src** when adding these new files.

DESCRIPTION

For this final task, you need to create the executor component for SPLAT, which takes an Abstract Syntax Tree (AST) produced by the parser and checked by the semantic analyzer, and executes it statement by statement. In the event that there is an error during execution, an ExecutionException should be thrown.

The class files that are provided for this task are:

- **Executor.java** – (Located in the **splat.executor** package.) This is the executor component that you will be implementing. Actually, much of the code for this is already provided – the constructor takes a program AST as its argument, and the runProgram() method does two things:
    - Creates the function map, which is needed when a function is called during program execution; and creates the variable map, which keeps track of the values for each of the variables declared for the program. You will have to implement the setMaps() method that does this, although it should be very similar to the setMaps() method in the SemanticAnalyzer.
    - Executes all of the statements in the program body, by calling the execute() method on each of them.
- **ExecutionException.java** – (Part of the **splat.executor** package.) Exceptions that might be thrown during execution. The only one that immediately comes to mind is a “divide by zero” error that might occur during division or a modulus operation. You shouldn’t have to change this file.
- **Value.java** – (Also part of the **splat.executor** package.) This class is needed to represent values that might be stored in variables or parameters, or might be returned by a function. This class is different than the Literal class(es), although it might be similar in the way it is implemented. You may also decide to create several subclasses of Value, to represent things of different types.

- **ReturnFromCall.java** – (Also part of the **splat.executor** package.) This exception class gives us an easy way to prematurely end the execution of a function when a return statement is reached, and return control to the function call Statement or Expression that initiated the function's execution. The idea is that when a return statement is executed within the context of a function body, it would throw a new ReturnFromCall exception, and could also include the return value (if necessary) as part of the exception. The original calling code (either a function call statement or expression) would then catch this type of exception, and potentially use the value bundled with the exception.

Main Tasks

Once you add the new package and files to your SPLAT project, you need to do the following:

1. Uncomment the final two phase 4 commands in your Splat class.
2. Complete the setMaps() method in your Executor class.
3. Uncomment the execute() and evaluate() methods in your Statement and Expression classes.
4. Implement the execute() and evaluate() methods for all of your Statement and Expression classes – most of your work for Phase 4 will focus on this.

The execute() and evaluate() methods are very similar to the analyze() and analyzeAndGetType() methods from the semantic analysis task, except that instead of determining types, here you are computing values and updating variables.

- **execute()** – This method essentially runs the given statement, which may result in changing variable or parameter values, printing values to the console window, executing loops, etc. You may also call a (void) function in a statement, in which case you will need to retrieve its declaration from the funcMap to set up a new varAndParamMap, and then proceed to execute the individual body statements of the function.
- **evaluate()** – This method calculates and returns the value of the given expression, which may require that you evaluate subexpressions, perform operations, etc. You may encounter a single variable or parameter label, in which case you may simply return its current value in the varAndParamMap. You may also call a (non-void) function in an expression, in which case you will need to retrieve its declaration from the funcMap to set up a new varAndParamMap, and then proceed to execute the individual body statements of the function.