Phase 3 – SPLAT Semantic Analyzer

Target Date: Friday, November 10

PHASE 3 SETUP

Starting with the same Java project from the previous tasks, add in the provided package folder, **splat.semanticanalyzer** found in the archive file semantic_start.zip, making sure to preserve the proper package structure in **src**. The two class files that are initially provided in this package are:

- **SemanticAnalyzer.java** – The semantic analyzer/typechecker that you will be implementing.
- **SemanticAnalysisException.java** – Exceptions that might be thrown during this phase. You shouldn't have to change this file.

Before you continue, don't forget to uncomment the two lines of code under "Step 3. Semantic Analysis" in Splat.java, which essentially connects the parser phase to this phase.

There is also one other small change that you should make to your declaration classes that simplifies the code in SemanticAnalyzer – specifically, you should move your Label fields and getters from FunctionDecl and VariableDecl into their parent class, Declaration. This will also require some small modifications to the constructors of all these classes. The reason for this change is to allow for easier access and lookup of the specific declarations that match labels that are found in statement code.

DESCRIPTION

For this task, you need to create the semantic analyzer component for SPLAT, which takes an Abstract Syntax Tree (AST) produced by a parser, and throws a SemanticAnalysisException if there are any typechecking or semantic errors in the AST. Note that this component does not produce anything that is used as input to the next phase, execution, but it is very important that your program passes this phase before you try running it.

Once you have set up your project for this phase, we are ready to start looking at the SemanticAnalyzer class in detail. The key method in the class is analyze(), which performs the following high-level tasks:

1. Makes sure that there are no duplicate labels among the variables and functions declared at the "global" program level. Although the method that does this is already implemented, you still should look over it carefully and understand how it works, since you will need to write a similar method that will be used for each of the functions.

2. Collects information about the functions and variables of the program, and stores them into maps for later use. This method is also provided for you, but it is worthwhile to look at it carefully to understand how it works.
3. Performs semantic analysis on each of the functions, which includes the subtasks:
    a. Checking for no duplicate labels among its parameters, local variables, and already existing function names.
    b. Storing type information for the parameters and variables (and return type) in a map.
    c. Perform semantic analysis on the body of the function by calling the analyze method on each of the statements. You will need to uncomment the analyze method in the abstract class Statement, and then implement this method in each of the subclasses of Statement. Similarly, you will need to uncomment the analyzeAndGetType method in the abstract class Expression, and implement it for each of its subclasses. Implementing these methods will likely be the most time-consuming part of this task.
4. Performs semantic analysis on the body of the program – i.e., its list of statements, by calling analyze on each of its statements.

Implementing analyze and analyzeAndGetType

As mentioned above, implementing these methods for each of your Statement and Expression classes should take the bulk of your effort for this task. The idea behind these methods are as follows:

- **analyzeAndGetType** – This method has two main purposes: (1) to perform semantic analysis on the expression, and (2) to return the actual type of the expression. Semantic analysis first needs to be done on its subexpressions by recursively calling analyzeAndGetType, and then the types returned by these calls can be used to make sure they properly match their expected types within the context of the larger expression.
- **analyze** – This method does basic semantic analysis on the given statement, calling analyzeAndGetType on expressions that make up the statement, getting their types, and performing typechecking when necessary.

Semantic rules that need to be followed and checked by these methods are outlined in the semantics.pdf file. Note that both methods take the arguments:

- Map<String, Type> varAndParamMap
- Map<String, FunctionDecl> funcMap

The variable and parameter map is used when we encounter an expression that is a label representing a variable or parameter. If the variable or parameter is not in the map, this should

result in a “variable not defined” error. If it is defined, then we can retrieve the type, and return it as the result of the analyzeAndGetType method.

Note that we can also use the variable and parameter map to store the return type of the current function. This is needed to properly typecheck return statements in function bodies. Consider storing this by using an illegal label name as the key, e.g., “0result” – this will prevent any possibilities of name clashes with any already existing variable or parameter names.

The function map is needed when a function call is found as an expression (in which case it needs to be non-void), or a function call is used as a statement (in which case it needs to be void.) In these cases, the function definition needs to be retrieved from the map in order to check the expected types of the parameters against the actual types of the arguments in the call, and if the return type is what it should be.