6.035 Project 1 Write-Up

Nicolaas Kaashoek, Endrias Kahssay, Tony Wang

1 Design

1.1 Internal Representation

We built our internal representation out of the `ParserRuleContexts` that ANLTR provides upon completing a parse. While we may have been able to use what ANTLR provides and do our semantic checking from there, we felt that building our own IR would make life much easier down the line.

We used Scala's case classes as the building blocks for our IR. Using case classes gives us the ability to do pattern matching on instances of a parent class (such as the different kinds of expressions). However, case classes are immutable, which may cause problems in future stages of the project. For now, we are sticking with case classes.

Each node on our IR tree is an instance of the `IRNode` trait, and contains some meta-data that allows us to print debugging information. ANTLR4 gives us the root of the tree in the form of a program context, which we then turn into a `ProgramNode` case class. From there, we iterate over all the import, field and method declarations and recursively construct the corresponding nodes on the tree. We then return the root of the tree.

After constructing the IR, we proceed to do semantic checking. We chose to separate semantic checking and the building of the IR to simplify the process. To start with, we represent the different scopes in a program with the `DecafSymbolTable` class. Each instance of the class also contains a pointer to a parent symbol table, which we use to maintain isolation between different blocks, as well as to enable shadowing. A look-up in the symbol table simply checks the current scope, and if nothing is found for that identifier, recursively checks parent scopes as well. As items are added to the symbol table we also do the checks of undeclared or redeclared identifiers. Additionally, as the symbol table is built up, we chain method declarations together in order to catch methods calling methods that haven't been declared yet. This is done using the parent pointer previously mentioned.

The next phase of semantic checking is the type checking. A program consists of many blocks of code within method declarations, and we iterate over these blocks, validating each statement in order. We wrote statement and expression validation code for every kind of statement/expression, and we use this as our type checking. After iterating through every block, we can be sure that all statements are correctly typed.

This leads to the final set of semantic checks, which we refer to as "one-offs". These checks are checking whether methods expected to return actually return, whether all continue/break statements are correctly inside a for/while loop, and the constraints on the main method. Of these checks, both the continue/break checks and the checks for returns could probably have been done at the same time as type checking. However, we chose to divide them into a separate set of checks for two reasons. First, it allowed us to better split up the work. Second, it greatly simplified the logic needed for type checking and made the debugging process much simpler. Given that our compiler still compiles in less than 20 seconds, we felt this was a fair trade-off.

We use the `SemanticException` object to keep track of any exceptions encountered during semantic checking. When an exception is found, we log the exception and continue as if that code didn't cause a problem. This allows us to catch all exceptions in one pass through, as opposed to quitting out on the first one found. The only exception to this is an undeclared variable, as we cannot reliably guess what the expected behaviour would be given such an exception. In this case we just error out.

1.2 Semantic Checking

2 Extras

First, we upgraded to using ANTLR4 instead of ANTLR2. This was done during the scanner/parser project, as it greatly improved the clarity of the grammar we ended up with, and allowed us to better create the internal representation used in this part of the project.

We modified the default ANTLR error handling strategy as well. The test cases provided by course staff check to see if our parser returns a non-zero exit code when an illegal parse is given. However, the default ANTLR way of doing this suppressed all error messages produced during the parse. To fix this, we wrote our own listener for errors that displays errors as well as exits with a non-zero exit code.

In order to better debug the internal representation we used in the project, we use the `json4s` library for JSON pickling in scala. Because JSON has become such a standard representation of objects in recent years, we determined that it would be a good way to debug the internal representation, as many external tools exist for pretty-printing JSON.

We didn't find any clear documentation on the expected behavior of local variables with the same name as parameters in a method. As a team, we decided not to allow local variables to shadow parameters, as it seems counter-intuitive from a programming standpoint.

Finally, we wrote our own unit testing framework based on the test cases provided by course staff. Using scalatest, we were able to greatly improve the speed at which the tests run, which made iterating our design much more efficient.