CS 6410: Compilers

Fall 2023

Part 3 – Static Semantics and Type Checking
Acknowledgment: Project assignment modified from an assignment developed by Hal Perkins, faculty member of the University of Washington, Allen School of Computer Science and Engineering

Posted on Wednesday, October 4, 2023
Instructor: Tamara Bonaci
Khoury College of Computer Science
Northeastern University – Seattle

Please submit your project by 11:59pm on Saturday, November 19, 20123. You should submit your project by pushing it to your Khoury GitHub repository, and providing a suitable tag. See the end of this writeup for details.

1 Assignment Overview

In this part of the project, your task is to add static semantics checking to your compiler. In particular, you should do the following:

– Add global symbol table(s) storing information about classes and their members, and local symbol tables for each method to store information about parameters and local variables.
– Compute and store type information for classes, class members, parameters, and variables.
– Compute type information for expressions and other appropriate parts of the abstract syntax.
– Add error checking to verify at least the following properties:
  1. Every name is properly declared.
  2. Components of expressions have appropriate types (e.g., + is only applied to values of type int, && is only applied to values of type boolean, the expression in parentheses that are a part of an `if` or `while` statement is boolean, etc.).
  3. If a method is selected from a value with a reference type, then that name is defined as a member of that type.
  4. Methods are called with the correct number of arguments.
  5. In assignment statements and method call parameter lists, the values being assigned have appropriate types (i.e., the value either has the same type as the variable or is a subclass of the variable's class).
  6. If a method in a subclass overrides one in any of its superclasses, the overriding method has the same parameter list as the original method, and the result type of the method is the same as the result type of the original one, or a subclass if the original result type is a reference type.
  7. There are no cycles in the inheritance graph - i.e., check to be sure that no class directly or indirectly extends itself.
– Feel free to add additional checks, but you should try to get this much done. It may be possible to do some of these checks on the fly while building symbol tables and type information, or it may require an additional visitor pass.
– If your parser grammar accepts input that is not a part of MiniJava (i.e., uses a covering grammar or otherwise successfully parses constructs that are syntactically legal but have semantic errors), check that the program is actually legal. If you implemented extensions like casts, you should add appropriate error and type checking.
– Print suitable messages describing any errors detected. It's fine to suppress useless error messages - for instance, feel free to complain only once about an undeclared variable instead of repeating the message each time the variable is used in the code.
– Your semantics checking, however, should generally continue to report multiple unrelated errors in a source program rather than stopping immediately after the first error is encountered. But feel free to put in some threshold if you want to terminate the compiler after it has produced a large number of error messages.

Modify your MiniJava main program so that when it is executed using the command

```
java MiniJava -T filename.java
```

it will parse the MiniJava program in the named input file, perform semantic checks as described above, and print the contents of the compiler symbol tables. We do not specify the detailed format of the symbol table output, but there should be one table for each scope, clearly labeled to identify the scope (class or method in most cases), and showing the names declared in that scope, their types, and any other important information. The output should not be any more verbose than necessary.

The `java` command shown above will also need a `-cp` argument or `CLASSPATH` variable as before to locate the compiled `.class` files and libraries. See the scanner assignment if you need a refresher on the details.

As with the previous parts of the compiler project, the compiler `main` method should return an exit or status code of 1 if any errors are detected in the source program being compiled (including errors detected by the scanner and parser, as well as semantics and type checks). If no errors are found, the compiler should terminate with a result code of 0.

Your MiniJava compiler should still be able to print out scanner tokens if the `-S` option is used instead of `-T`, and `-P` or `-A` should continue to print the AST in the requested format. There is no requirement for how your compiler should behave if more than one of `-A, -P, -S` and `-T` are specified at the same time. That is up to you. You could treat that as an error, or, maybe more useful, the compiler could print an appropriate combination of tokens, trees, and symbol tables.

As before, If you are using a different implementation language or additional libraries, please be sure that your compiler continues to work as similarly as possible to the specification above, and you must add to your README file any new or additional information we need to build, run, and test your compiler.

2 Details and Suggestions

Now would be a good time to go back and re-read the MiniJava project overview, and recheck the language grammar to remind yourself of what is, and is not included in the MiniJava subset of Java. Your compiler may, of course, contain extensions to MiniJava, but be sure to refresh your memory about what is contained in the core language.

It's probably easiest to collect the type information in multiple passes over the AST. An initial pass should collect information about classes and fields (both data and methods), and build the global symbol tables. A later pass would then analyze method bodies, build the local symbol tables, and perform type, and other error checking. You might find it more convenient to break this down into more passes, each of which does fewer things, particularly for the initial pass, where it might be easier to build a global symbol table of class names before processing individual classes to build class symbol tables with information about variables, methods, and their types.

You should add appropriate fields in some or all AST nodes to store references to type and other information as necessary. But remember that you should have a separate data abstraction (ADT) to represent type information used for semantics checking in the compiler, and not confuse this information with the source program type declarations in the AST.

Remember the visitor pattern that we used for the parser part. Please make use of it! This is where it pays off to have gone to the trouble to set up the visitor machinery. Provide new implementations of the Visitor interface as needed to do the semantics checks.

Take advantage of the standard library container classes and data structures in Java to simplify your implementation. Class HashMap should be particularly useful for symbol tables. Use the List classes (ArrayList or LinkedList) for things like argument and parameter lists. Don't reinvent any more wheels than necessary.

It can be useful to include a few auxiliary methods that perform common operations on types. Possibilities include a method that returns true if two types are the same, and a method that returns true if a value of one type is assignable to another. Also possibly useful: a method that tries to add an entry to a symbol table and reports an error if the name is already declared, and another that looks up an identifier, and reports an error if it is not found (and maybe adds it to the symbol table with an "undefined" type, which can be used to suppress additional redundant error messages about the same identifier).

You should test your compiler by processing several MiniJava programs, both correct ones and ones with errors. Be sure to check some examples that are syntactically legal (i.e., can be parsed with no errors) but contain semantic errors.

You should continue to use your Khoury GitHab repository to store the code for this, and remaining parts of the compiler project.

3 What to Submit

For this phase of the project, I will be looking to see if your compiler properly performs at least the semantics checks listed in the Overview section above. I will also check if it can print the requested symbol tables, and information in a reasonable format. Additionally, I will check whether your compiler can handle MiniJava programs containing errors, as well as ones that are legal.

You should include a brief `SEMANTICS-NOTES` file describing any additional checks or other extensions you included in this phase of the compiler. You should also give a brief explanation of any changes you needed to make in previous parts of the project (scanner, parser, ASTs) as you implemented the semantics checks.

As before, you will submit this part of the project by pushing code to your GitHub repository. Once you are satisfied that everything is working properly, create a `semantics-final` tag, and push that to the repository.