Programming Assignment

Due Wednesday, September 20th, 2017 at 11.59 PM

1 Introduction

In this assignment, you will implement the static semantics of Cool. You will use the abstract syntax trees (AST) built by the parser to check that a program conforms to the Cool specification. Your static semantic component should reject erroneous programs; for correct programs, it must gather certain information for use by the code generator. The output of the semantic analyzer will be an annotated AST for use by the code generator.

This assignment has much more room for design decisions than previous assignments. Your program is correct if it checks programs against the specification. There is no one "right" way to do the assignment, but there are wrong ways. There are a number of standard practices that we think make life easier, and we will try to convey them to you. However, what you do is largely up to you. Whatever you decide to do, be prepared to justify and explain your solution.

You will need to refer to the typing rules, identifier scoping rules, and other restrictions of Cool as defined in the Cool Reference Manual. You may also need to add methods and data members to the AST class definitions for this phase.

There is a lot of information in this handout, and you need to know most of it to write a working semantic analyzer. *Please read the handout thoroughly.* At a high level, your semantic checker will have to perform the following major tasks:

1. Look at all classes and build an inheritance graph.
2. Check that the graph is well-formed.
3. For each class
   (a) Traverse the AST, gathering all visible declarations in a symbol table.
   (b) Check each expression for type correctness.
   (c) Annotate the AST with types.

This list of tasks is not exhaustive; it is up to you to faithfully implement the specification in the manual.

For this assignment, you can assume that the input programs do **not** have any `SELF_TYPE` type. This assumption leads to *substantial* reduction of the code that you have to write making the assignment doable in around 2 weeks.

CS3423 registrants must work in a group for this assignment, where a group consists of two students; in special circumstances, three per team may be allowed. For CS6240 registrants, it is a choice whether they choose to do it individually or in a team of two.

2 Files and Directories

To get started, create a directory where you want to do the assignment and extract the given **semantic.tar.gz** in it.

We now describe the most important files that you may want to modify for your project.

- src/java/cool/Semantic.java
  This is the main file for your implementation of the semantic analysis phase. The semantic analyzer is invoked by the constructor of class Semantic. You are given the top level program node of the AST through which you will need to traverse the AST.

- java/cool/ScopeTable.java
  This contains the implementation of a generic Scope Table. You are free to use this or implement your own Scope Table.

- java/cool/AST.java
  This contains the implementation of the AST used in the parsing stage. You can modify this file for adding member functions in nodes.

- src/test_cases/*.cl
  These files test a few semantic features. You should add tests to ensure that good.cl exercises as many legal semantic combinations as possible and that bad.cl exercises as many kinds of semantic errors as possible. It is not possible to exercise all possible combinations in one file; you are only responsible for achieving reasonable coverage. Explain your tests in these files and put any overall comments in the README file.

- README
  This file will contain the write-up for your assignment. For this assignment, it is critical that you explain design decisions, how your code is structured, and why you believe that the design is a good one (i.e., why it leads to a correct and robust program). It is part of the assignment to explain things in text, as well as to comment your code. Inadequate README files will be penalized more heavily in this assignment, as the README is the major guideline we have to understanding your code.

3 Tree Traversal

Your programming task for this assignment is to (1) traverse the tree, (2) manage various pieces of information that you glean from the tree, and (3) use that information to enforce the semantics of Cool. One traversal of the AST is called a "pass". You will probably need to make at least two passes over the AST to check everything.

You will most likely need to attach customized information to the AST nodes. To do so, you may edit AST.java directly.

4 Inheritance

Inheritance relationships specify a directed graph of class dependencies. A typical requirement of most languages with inheritance is that the inheritance graph be acyclic. It is up to your semantic checker

to enforce this requirement. One fairly easy way to do this is to construct a representation of the type graph and then check for cycles.

In addition, Cool has restrictions on inheriting from the basic classes (see the manual). It is also an error if class A inherits from class B but class B is not defined.

You will need to add appropriate definitions of all the basic classes and incorporate these classes into the inheritance hierarchy.

We suggest that you divide your semantic analysis phase into two smaller components. First, check that the inheritance graph is well-defined, meaning that all the restrictions on inheritance are satisfied. If the inheritance graph is not well-defined, it is acceptable to abort compilation (after printing appropriate error messages, of course!). Second, check all the other semantic conditions. It is much easier to implement this second component if one knows the inheritance graph and that it is legal.

5 Naming and Scoping

A major portion of any semantic checker is the management of names. The specific problem is determining which declaration is in effect for each use of an identifier, especially when names can be reused. For example, if **i** is declared in two let expressions, one nested within the other, then wherever **i** is referenced the semantics of the language specify which declaration is in effect. It is the job of the semantic checker to keep track of which declaration a name refers to.

As discussed in class, a *symbol table* is a convenient data structure for managing names and scoping. You may use our implementation of symbol tables for your project. Our implementation provides methods for entering, exiting, and looking up as needed. You are also free to implement your own symbol table, of course.

Besides the identifier **self**, which is implicitly bound in every class, there are four ways that an object name can be introduced in Cool:

- attribute definitions;
- formal parameters of methods;
- let expressions;
- branches of case statements.

In addition to object names, there are also method names and class names. It is an error to use any name that has no matching declaration. In this case, however, the semantic analyzer should *not* abort compilation after discovering such an error. Remember that neither classes, methods, nor attributes need be declared before use. Think about how this affects your analysis.

6 Type Checking

Type checking is another major function of the semantic analyzer. The semantic analyzer must check that valid types are declared where required. For example, the return types of methods must be declared. Using this information, the semantic analyzer must also verify that every expression has a valid type according to the type rules. The type rules are discussed in detail in the Cool Reference Manual and the course lecture notes. Also remember that you need not handle the `SELF_TYPE` type.

One difficult issue is what to do if an expression doesn't have a valid type according to the rules. First, an error message should be printed with the line number and a description of what went wrong. It

is relatively easy to give informative error messages in the semantic analysis phase, because it is generally obvious what the error is. We expect you to give informative error messages. Second, the semantic analyzer should attempt to recover and continue. We do expect your semantic analyzer to recover, but we do not expect it to avoid cascading errors. A simple recovery mechanism is to assign the type Object to any expression that cannot otherwise be given a type (we used this method in coolc).

7 Code Generator Interface

For the semantic analyzer to work correctly with the rest of the coolc compiler, some care must be taken to adhere to the interface with the code generator. We have deliberately adopted a very simple, naïve interface to avoid cramping your creative impulses in semantic analysis. However, there is one thing you must do. For every expression node, its type field must be set to the Symbol naming the type inferred by your type checker. The special expression no_expr must be assigned the type No_type.

8 Expected Output

For incorrect programs, the output of semantic analysis is error messages. You are expected to recover from all errors except for ill-formed class hierarchies. You are also expected to produce complete and informative errors. Assuming the inheritance hierarchy is well-formed, the semantic checker should catch and report all semantic errors in the program. Your error messages need not be identical to those of **coolc**.

We have supplied you with simple error reporting methods **reportError()**. This routine takes a filename, line number and the error message string. You can obtain the line number from the AST node where the error is detected and the filename from the enclosing class.

For correct programs, the output is a type-annotated abstract syntax tree. You will be graded on whether your semantic phase correctly annotates ASTs with types.

9 Testing the Semantic Analyzer

You will need a working scanner and parser to test your semantic analyzer which we have provided. You will run your semantic analyzer using semantic, a shell script that "glues" together the analyzer with the parser and the scanner.

10 Remarks

The semantic analysis phase is by far the largest component of the compiler so far. You will find the assignment easier if you take some time to design the semantic checker prior to coding. Ask yourself:

- What requirements do I need to check?
- When do I need to check a requirement?
- When is the information needed to check a requirement generated?
- Where is the information I need to check a requirement?

If you can answer these questions for each aspect of Cool, implementing a solution should be straightforward.

11 Final Submission

Make sure to complete the following items before submitting to avoid any penalties.

- Include your write-up in README.
- Include your test cases that should pass the semantic analyzer and those that should cause the semantic analyzer to issue an error.
- Make sure to include all the files you changed for the semantic analyzer