Phase III: The Semantic Checker

Due Date: November 16th.
Teamwork: Highly encouraged.

Purpose:

This project is intended to give you experience in and bring together the various issues of semantic error checking, type checking, and symbol table manipulation discussed in class. In performing these tasks for the Cool language, you will be performing abstract syntax tree traversals and dealing with the inheritance hierarchy of an object-oriented language.

Groupwork:

The same rules for group work as in Phase II apply.

Project Summary:

Your task is to write a semantic analysis phase for your Cool compiler. In effect, you are implementing the static semantics of Cool. You will use the abstract syntax trees (AST) built by the parser to check that a program is in conformance with 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 attributed 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 "right" way to do the assignment, but there are wrong ways. There are a number of standard practices which 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-manual. You will 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.*

Files:

You already have all necessary files in your working copy. Make sure all files are up-to-date, and make sure you get the README for this phase.

The java source files that you will need to modify are:

- treeNodes/*.java
  This file contains the definitions for the AST nodes. You will need to add the code for your semantic analysis phase in this file. The semantic analyzer is invoked by calling method **semant()** of class **program**. Do not modify the existing declarations.

- semanticAnalyzer/ClassTable.java
  This class is a placeholder for some useful methods (including error reporting and initialization of basic classes). You may wish to enhance it for use in your analyzer.

Do not modify any other given java source files. You are allowed to create new java classes as you wish.

Make sure that you understand the use of all the classes *semanticAnlyzer/*.java* and *generalHelpers/*.java*

Tasks:

1. Read Section 12 in the Cool-Manual. This section defines all the type checking rules for Cool. Understanding Section 12 is vital for working on this assignment.

2. Modify the classes treeNodes/*.java and semanticAnalyzer/ClassTable.java as needed to annotate a given AST with types according to the typing rules given in Section 12 of the Cool-Manual. Reject ASTs representing semantically incorrect programs.

3. As usual, the file README_Phase3.txt contains detailed instructions for the assignment. You should also edit this file to include the write-up for your project. Explain your design decisions and why you believe your program is correct and robust. It is part of the assignment to explain things in text, as well as to comment your code.

4. Test and debug your code. The readme gives instructions as to how to debug your code (note that there is a new directory at the root of the repository containing specific files to check your semantic analyzer with).

AST Traverals:

As a result of Phase II, your parser builds abstract syntax trees. The method **dump_with_types**, defined on most AST nodes, illustrates how to traverse the AST and gather information from it. This algorithmic style—a recursive traversal of a complex tree structure—is very important, because it is a very natural way to structure many computations on ASTs.

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.

As an example approach, the coolc compiler performs three passes as follows:

Pass 1: This is not a true pass, as only the classes are inspected. The inheritance graph is built and checked for errors. There are two "sub"-passes: check that classes are not redefined and inherit only from defined classes, and check for cycles in the inheritance graph. Compilation is halted if an error is detected between the sub-passes.

Pass 2: Symbol tables are built for each class. This step is done separately because methods and attributes have global scope—therefore, bindings for all methods and attributes must be known before type checking can be done.

Pass 3: The inheritance graph—which is known to be a tree if there are no cycles—is traversed again, starting from the root class Object. For each class, each attribute and method is typechecked. Simultaneously, identifiers are checked for correct definition/use and for multiple definitions. An invariant is maintained that all parents of a class are checked before a class is checked.

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

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.

The project skeleton includes appropriate definitions of all the basic classes. You will need to 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.

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 augmenting scopes 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, of course, an error to use any name that has no matching declaration.

Remember that neither classes, methods, nor attributes need be declared before use. Think about how this affects your analysis.

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-manual and the course lecture notes.

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. A good semantic analyzer will avoid cascading errors using any of several standard techniques. 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).

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. This Symbol must be the result of the addString (Java) method of the idtable. The special expression no_expr must be assigned the type No_type which is a predefined symbol in the project skeleton.

Output and Grading:

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. When in doubt, use **coolc** as a guide in determining what informative error messages should say. Your error messages need not be identical to those of **coolc**.

We have supplied you with a simple Exception-subclass **semanticAnalyzer/SematicError.java**. This class has several constructors for your convenience and a new instance of this class should be created when seeing finding an error (it is up to you, whether you would like to throw it). The filename to be specified should be the file in which the error occurs. The parser ensures that Class_ nodes store the file in which the class was defined (recall that class definitions cannot be split across files). In an error message, the line number of the error message is obtained from the AST node where the error is detected and the file name is obtained 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 and on whether your semantic phase works correctly with the `coolc` code generator.

You are also expected to program in good, structured style. You should spend some time thinking about the class definitions you will use.

Many students have problems with properly handling issues with "SELF_TYPE". 5% of the grade are dependent only on whether SELF_TYPE has been handled properly, and in no other place will deductions be made, if some problem with SELF_TYPE arises.

Remarks:

The semantic analysis phase is by far the largest component of the compiler so far. Our solution is approximately 1300 lines of well-documented C++. 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. 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.

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

Turn in: Your project should be submitted as usual to the svn-repository.

Modifying java-source files differently from the ones stated in this assignment need either prior approval by the TA or will result in penalty. You may of course do any changes you wish for your private use, as long as this does not affect your submission.