Programming Assignment 2: “Semantic Analysis”

1 Overview

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 the previous assignment. Your program is correct if it checks programs against the specification. There is no one “right” way to do this assignment, but 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.

The code templates we provide are described in this handout. You will need to refer to the typing rules, identifier scoping rules, and other restrictions of Cool as defined in the Cool Reference Manual.

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.

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.*

You must work on this assignment in the same team as for A1.

1.1 AST Traversal

As a result of A1, your parser builds abstract syntax trees. The class `DumpVisitor` from the package `ast/visitor` illustrates how to traverse the AST and gather information from it. This algorithmic style—a recursive traversal of a complex tree structure—is important, because it is a natural way to structure many computations that work 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.

You will most likely need to attach customized information to the AST nodes. You must not change the classes in `ast` package. You can use the field `type` of class `ExpressionNode`. If you need to record any other information associated with AST nodes, create another data structure, mapping AST nodes to the information that you need to record.

1.2 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 some basic classes (see the manual). It is also an error if class A inherits from class B but class B is not defined.

The code template 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). 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.

1.3 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.

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

1.4 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 in class.

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. Second, the semantic analyzer should attempt to recover and continue. A simple recovery mechanism is to assign the type Object to any expression that cannot otherwise be given a type (this method is used in coolc).

1.5 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 method of the idtable. The special expression NoExpressionNode must be assigned the type No_type which is a predefined symbol in TreeConstants.java.

1.6 Expected Output

For incorrect programs, the output of semantic analysis is error messages. You are 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`.

Code template includes a simple error reporting method `PrintStream Utilities.semantError(ClassNode)`. This method takes a ClassNode and returns an output stream that you can use to write error messages. Since the parser ensures that ClassNode objects store the file in which the class was defined (recall that class definitions cannot be split across files), the line number of the error message can be obtained 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 marked on whether your semantic analysis correctly annotates ASTs with types and on whether your semantic phase works correctly with the `coolc` code generator.

1.7 Symbol Tables

Compilers must also determine and manage the scope of program names. A symbol table is a data structure for managing scope. Conceptually, a symbol table is just another lookup table. The key is the symbol (the name) and the result is whatever information has been associated with that symbol (e.g., the symbol’s type).

In addition to adding and removing symbols, symbol tables also support operations for entering and exiting scopes and for checking whether an identifier is already defined in the current scope. The lookup operation must also observe the scoping rules of the language; if there are multiple definitions of identifier `x`, the scoping rules determine which definition a lookup of `x` returns. In most languages, including Cool, inner definitions hide outer definitions. Thus, a lookup on `x` returns the definition of `x` from the innermost scope with a definition of `x`.

Cool symbol tables are implemented as lists of scopes, where each scope is a hash table of $\langle \text{identifier}, \text{data} \rangle$ pairs. The “data” is whatever data the programmer wishes to associate with each identifier.

1.8 Designing the Semantic Analyzer

You will find this 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 straight-forward.

1.9 Testing the Semantic Analyzer

You will need a working scanner and parser to test your semantic analyzer. You may use either your own scanner/parser or the reference scanner/parser. By default, the reference components are used. We will mark your semantic analyzer using the reference scanner and parser.

You can run your semantic analyzer using `mysemant`, a shell script that “glues” together the analyzer with the parser and the scanner. Note that `mysemant` takes a `-s` flag for debugging the analyzer; using this flag merely causes a static field of the class `Flags` to be set. Adding the actual code to produce useful debugging information is up to you. See the project `README` for details.

Once you are confident that your semantic analyzer is working, try to invoke your analyzer together with other compiler phases. You should test this compiler on both good and bad inputs to see if everything is working. Remember, bugs in the semantic analyzer may manifest themselves in the code generated or only when the compiled program is executed under `spim`.

2 Getting and Submitting the Assignment

2.1 Initial Setup

1. This assignment must be carried out in a new `semant` branch in your existing group repository. **One team member** should do:

   ```
   cd $COOL_HOME
   git checkout main
   git checkout -b semant
   git push -u origin semant
   ```

   Then, the **other team members** should do:

   ```
   cd $COOL_HOME
   git pull
   git checkout semant
   git push -u origin semant
   ```

   Note, this new `semant` branch is based off the *original* `main` branch, i.e., not your `frontend` branch.

2. One person should add the provided `ASTLexer.g4` and `ASTParser.g4` files to a new directory `src/ast/parser` in the new branch and push them to the group repo. The other group members should pull these files from the group repo.

3. Before proceeding further, check that the new branch is visible and correct for all team members.

2.2 A2 Instructions and Submission

1. To compile and run the template code, type

   ```
   cd $COOL_HOME/assignments/pa2
   buildme semant
   ./mysemant good.cl
   ```

   Try it straight away—it should "work", and print back the same AST (because nothing is implemented yet).

2. Check the instructions in the `assignments/pa2/README` file.

3. During your work, you can compare your output against that of the reference compiler by running `./refsemant` as opposed to `./mysemant`. Make your output to match the reference version to pass all the testme tests.

4. The files that you may modify are:

   - `Semant.java`
   - `ScopeCheckingVisitor.java`
   - `TypeCheckingVisitor.java`
   - `ClassTable.java`
   - `TreeConstants.java`

   You should NOT edit any other files nor add any other files that are required to compile your project. In fact, if you modify any other files, you may find it impossible to complete the assignment.

5. Again, you will submit a zip archive of your entire `ECS652U-cw-group<number>` code repository, including the (possibly "hidden") `.git` directory.

   Failure to include the `.git` directory will count as an incomplete submission. To reiterate: we will mark your project based on the latest commit on the `semant` branch (regardless of the commit message) at the deadline.