

# Programming Assignment III

Due Thursday, May 15, 2025 at 11:59pm

## 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.

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 will also need to add methods and data members to the AST class definitions for this phase. The functions the tree package provides are documented in the *Tour of Cool Support Code*.

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. As before, you may work individually or as a pair.

## 2 Files and Directories

To get started, log in to one of the *myth* machines and run one of the following commands:

```
/afs/ir/class/cs143/bin/pa_fetch PA3 <project_directory>
```

The project directory you specify will be created if necessary, and will contain a few files for you to edit and a bunch of symbolic links for things you should not be editing. (In fact, if you make and modify private copies of these files, you may find it impossible to complete the assignment.) See the instructions in the README file.

Here is a list of the files that you may want to modify:

- `cool-tree.h`

This file is where user-defined extensions to the abstract syntax tree nodes are placed. You will likely need to add additional declarations, but do **not** modify the existing declarations.

- `semant.cc`

This is the main file for your implementation of the semantic analysis phase. It contains some symbols predefined for your convenience and a start to a **ClassTable** implementation for representing the inheritance graph. You may choose to use or ignore these.

The semantic analyzer is invoked by calling method `semant()` of class `program_class`. The class declaration for `program_class` is in `cool-tree.h`. Any method declarations you add to `cool-tree.h` should be implemented in this file.

- `semant.h`

This file is the header file for `semant.cc`. You add any additional declarations you need (not in `cool-tree.h`) here.

- `good.cl` and `bad.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.

- `README`

This file contains detailed instructions for the assignment as well as a number of useful tips. You should edit this file to include your SUNet ID(s), just as you did in the first two programming assignments.

### 3 Tree Traversal

As a result of Programming Assignment 2, 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.

You will most likely need to attach customized information to the AST nodes. To do so, you may edit `cool-tree.h` directly. The method implementations you wish to add should go into `semant.cc`.

### 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.

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.

## 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 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 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.

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.

## 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. This `Symbol` must be the result of the `add_string` 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.

## 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. If you can, it will help us with grading if you match the text of the sample binaries' error messages. However, this is not required; your error messages need not be identical to those of `coolc` and do not need to be in the same order as the sample binaries to receive full credit.

We have supplied you with a simple error reporting method:

```
ostream&ClassTable :: semant_error(Class_)
```

This routine takes a `Class_` node and returns an output stream that you can use to write error messages. Since the parser ensures that `Class_` nodes 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 graded on whether your semantic phase correctly annotates ASTs with types and on whether your semantic phase works correctly with the `coolc` code generator.

## 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 `coolc` scanner/parser. By default, the `coolc` phases are used; to change that, replace the `lexer` and/or `parser` executable (which are symbolic links in your project directory) with your own scanner/parser. Even if you use your own scanner and/or parser, it is wise to test your semantic analyzer with the `coolc` scanner and parser at least once, because we will grade your semantic analyzer using `coolc`'s scanner and parser.

You will 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 the global variable `semant_debug` 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 running `mycoolc` 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`.

We provide a reference semantic analyzer for comparison. The reference can be found at `/afs/ir/class/cs143/bin/semant`.

## 10 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?

Once you can answer these questions for each aspect of Cool, implementing a solution will become easier.

## 11 Final Submission

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

- Update the README to include you and your partner's SUNet IDs, in the same manner as the previous programming assignments.
- Include sample Cool code that should pass the semantic analyzer in `good.cl`, and sample Cool code that should cause the semantic analyzer to issue an error in `bad.cl`. We also recommend that you add additional files to achieve higher test coverage, as it's not possible to demonstrate all the functionality of your semantic analyzer in just two files. However, we will not be grading `good.cl`, `bad.cl`, or any other test files you write.
- Make sure all your code for the semantic analyzer is in `cool-tree.h` (and/or `cool-tree.handcode.h`), `semant.h`, and `semant.cc`

To submit your assignment, run `make submit` from your PA3 directory and follow the instructions.