Programming Assignment IV

Due Date: Monday, November 6, 2006 (dated midnight).

Groups: You are permitted to work collaboratively as a group of 2 or 3 members. If you choose to work in a group, your group will be assigned an overall group pre-grade based on the grading of the assignment. Each member will complete a peer review, and your actual individual grade on the project will be a percentage of the group pre-grade based on how well your group worked together and your individual role in the work. If everyone worked well together in your group and participated fairly equally, you will all receive 100% of the group grade. If a particular member did not work near as much as the others, then their assigned grade for the project will be a percent lower than 100%.

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.

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 CoolAid. 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* and in the header files `cool-tree.h` and `tree.h` (C++ version). and online javadoc (Java Version).

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 and Directories:

To get started, create a directory where you want to do the assignment and execute one of the following commands *in that directory.* For the C++ version of the assignment, you should type

```
gmake -f ~pollock/public/cool02/assignments/PA4/Makefile
```

For Java, type:

```
gmake -f ~pollock/public/cool02/assignments/PA4J/Makefile
```

(notice the "J" in the path name). This command will copy a number of files to your directory. Some of the files will be copied read-only (using symbolic links). You should not edit these files. 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. The files that you will need to modify:

- semant.cc (C++ version)
  This file contains a start on a semantic analysis phase, written in C++. Put the code for your semantic analysis phase in this file. The skeleton only includes things required to correctly meet the interface with the code generator. (This interface is discussed in detail below.) Very little of real interest is included. The semantic analyzer is invoked by calling method **semant()** of class **program**. Unlike previous assignments, this skeleton does not even compile!

- semant.h (C++ version)
  This file is the header file for semant.cc.

- cool-tree.handcode.h and cool-tree.h (C++ version)
  These files are where user-defined extensions to the abstract syntax tree nodes are placed. You can modify either file, but you will probably find it easiest to modify cool-tree.h. You may add new `#define` statements, but do not modify the existing declarations, except for the class_EXTRA macros. You may add nay fields you wish to the class_EXTRA macros.

- symtab.h (C++ version)
  This file contains code for a simple symbol table module. You are not required to modify these files, but you are free to do so if the symbol table manager does not meet your needs. For example, you may wish to change the **SymtabEntry** template in symtab.h to add information to the symbol table. Document any changes you make to the original code.

- cool-tree.java (Java version)
  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.

- ClassTable.java (Java version)
  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.

- TreeConstants.java (Java version)
  This file defines some useful symbol constants.

- good.cl 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. 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.

Make sure that the name of each group member is in the README file.

As usual, there are other files used in the assignment that are symbolically linked to your directory or are included from ~pollock/public/cool02/include/PA4. You should not modify these files. Almost all of these files have have been described in previous assignments. The exceptions are ast.flex and ast.y, which implement lexical analysis and parsing for a textual representation of Cool ASTs. Recall that there are two versions of coolc: a normal executable and a shell script that glues separate parser, semantic analysis, and code generation phases together via pipes. Each phase uses dumptype to print the AST to the pipe, and the next phase uses the AST parser to reconstruct the tree data structure.[1]

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 `semant_debug` (a global variable in the C++ version and a static field of class `Flags` in the Java version) 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.

AST Traverals:

As a result of assignment 3, 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.

[1]One may wonder: Why have two versions of coolc? The "phased" version of coolc greatly simplifies the structure of the programming assignments by removing the need to guarantee that your code for one phase links with all components for other phases of the course compiler.

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 cool-tree.h (C++) or cool-tree.java (Java) directly. In the C++ version, any method definitions you wish to add should go into semant.cc.

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 CoolAid 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 add_string (C++) or 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 error reporting method **ClassTable::semant_error()** (C++) and **ClassTable.semantError()** (Java). This routine takes a filename and the AST node where the error occurred, and it returns the error stream after it has printed an error header. The filename 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.

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: Please follow the same submissions instructions as PA3.