Programming Assignment IV[1]

A Semantic Analyzer for COOL

PART A: Due Monday, October 31, 2014

PART B: Due Monday, November 10, 2014

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 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* 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 online in:

http://ece.uprm.edu/~bvelez/courses/Fall2014/icom4029/cooldoc/

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.

2. Files and Directories

Begin by creating a new directory for your **`PA4J`** code called **`MyPA4J`**. Then copy the original **`Makefile`** from the **`PA4J`** directory to the new directory:

- `cd ~/cool/assignments`
- `mkdir MyPA4J`
- `cp PA4J/Makefile MyPA4J`

Editing the **`Makefile`** in the **`MyPA4J`** directory to point the **`CLASSDIR`** variable to ~/cool or the directory where you unpacked the Cool source distribution.

To get the project files, type:

- `cd MyPA4J`
- `gmake -f Makefile source`

[1] This programming assignment is essentially the same one used at UC Berkeley CS 164 in the Spring of 2003. Used with permission.

(notice the \J" in the path name). This will create several symbolic links in the correct directory. You should not edit the files pointed to by these links. 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.

By this time you should have an Eclipse project with your Cool compile source code for PA2 and PA3. You should have tagged your PA2 and PA3 appropriately in order to be able to pull the version of each PA submitted for grading. Any changes you make to your project at this point will not affect any previously tagged versions of your project.

Open the **MyPA4J** directory using a file explorer in Linux and copy any Java source files not already existing in the Eclipse project. YOU SHOULD NOT COPY THE COOL-TREE.JAVA FILE. This file was renamed in PA3 to avoid several naming conflicts generated by Eclipse. You do not have to copy any files already included in your project.

All your Java source files should compile correctly, except for Semant.java. The compiler errors on this file are cause by references to a **Program** class that was refactored to **ProgramAbstract** in PA3. Proceed to rename these references accordingly. All your source code should compile correctly. We recommend that you tag your project at this time in order to be able to rollback your code to its "untouched" state for PA4 in case you have to.

Copy files **good.cl** and **bad.cl** to the root of your eclipse project.

This is a list of the files that you will need to modify:

- `CoolTree.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.

3. 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. If you run the **Semant.java** application you would need to paste parser AST output into the Eclipse console when you run the application.

To use your own lexer and parser replace the default lexer and/or parser classes in **Semant.java** (**ASTLexer** and **ASTParser**) with your own scanner/parser classes; normally **CoolLexer** and **CoolCup** respectively. To test your code paste actual Cool source code into the Eclipse console or add a filen name as a command line argument to the run configuration as for previous PA's.

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 running Semant.java as a Java applciation. Note that `Semant` takes a -s flag for debugging the analyzer; using this flag merely causes `semant_debug` (a static field of 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 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`.

4. Tree Traversal

As a result of assignment 3, your parser builds abstract syntax trees. The method **dumpWithTypes()**, 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 **CoolTree.java** directly.

5. 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. A well known method for accomplishing this is to conduct a depth-first traversal of the graph.

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.

6. 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.

7. 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`).

8. 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, naive 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 `no_expr` must be assigned the type `No_type` which is a predefined symbol in the project skeleton.

9. 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.semantError()`. 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, on whether your semantic phase works correctly with the coolc code generator, and on whether your semantic phase recovers from errors except for ill-formed class hierarchies.

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

Due to the size of this assignment and to facilitate its development, it will be divided into two parts:

PART A

i. Deliverables

On the first part your program is expected to do the following:

- Build a class inheritance graph or tree.
- Check that the graph is well-formed. (It is not cyclic).
- Check the following elements for type correctness and change their type accordingly:
  - Class Feature ‘attr’

- Class Method 'method'
- Formal Parameter 'formal'
- Integer Constant 'int_const'
- Boolean Constant 'bool_const'
- String Constant 'string_const'
- Assignment 'assign'
- Addition 'plus'
- Subtraction 'sub'
- Multiplication 'mul'
- Division 'divide'
- Instantiation 'new_'

- Perform error checking pertaining to the above expressions.

Ignore any other expressions that are not in this part (do not bother evaluating them) unless your implementation requires it, in which case you may return the "no_type" Symbol. The test cases will not include any expressions that are not of **PART A**. At this stage, your semantic analyzer may not yet work with the reference code generator.

ii. Turning In PART A

You must perform the following instructions for turning it in:

1. Make sure your code is in `CoolTree.java`, `ClassTable.java`, and `TreeConstants.java` and that it compiles and works.
2. Your test cases should be in `good.cl` and `bad.cl`.
3. Include any relevant comments in the `README` file and answer any questions that appear in it.
4. Make sure everything (`CoolTree.java`, `ClassTable.java`, `TreeConstants.java`, `good.cl`, `bad.cl`, and `README`) is updated on your GitHub repository.
5. Tag your GitHub project with label **PA4A** and send your tag label to the professor for grading.
6. Your code should be tagged before **October 31st, 2014 11:59pm**):

PART B

i. Deliverables

On the second part your program is expected to do the following:

- Check the following elements for type correctness and change their type accordingly:
  - Case 'typ_case'
  - Static Dispatch 'static_dispatch'
  - Dispatch 'dispatch'
  - If-Then-Else-Fi 'cond'
  - While-Loop-Pool 'loop'
  - Block 'block'
  - Let 'let'
  - Negation 'neg'
  - Less than 'lt'
  - Equality 'eq'
  - Less than or equals 'leq'

- Complement 'comp'
- Void Test 'isvoid'
- Object Variable 'object'
  (NOT the class type)

- Complete error checking.
- Correctly perform semantic analysis on any Cool program.
- Produce a type-annotated abstract syntax tree.
- Work with the other reference stages of the compiler.

ii. Turning In PART B

Follow the same steps as **PART A** above except for steps 5-6:

5. Tag your GitHub project with label **PA4A** and send your tag label to the professor for grading.
6. Your code should be tagged before **November 10, 2014 11:59pm**):

10. Notes

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.