Cairo University
Faculty of Computers & Information
Computer Science Department
Year 2014 – 2015
Second Term

Assignment-3: Semantic Analyzer

Based on Compilers Assignments at Stanford University

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. 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 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 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.
You must work in teams of atmost 3 members.

2. Files and Directories

To get started, create a directory where you want to do the assignment and execute the following commands in that directory:

```
make -f /usr/class/cs143/assignments/PA4/Makefile
```

As usual, there are several files used in the assignment that are symbolically linked to your directory or are included from `/usr/class/cs143/include/PA4`. Do not modify these files. Almost all of these files have have been described in previous assignments. See the instructions in the README file.

This 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.
  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. Feel free to modify these files to test your semantic analyzer.

- `README`
  This file contains detailed instructions for the assignment as well as a number of useful tips.

3. Tree Traversal

As a result of 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 (Implementation of `dump_with_types` is located in `dumptype.cc`. 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. Naming and Scoping

A major portion of any semantic checker is the management of names. 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 the provided implementation of symbol tables for your project. The provided implementation supports 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.

5. 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.

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 (this method is used in coolc).

6. Code Generator Interface

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.

7. 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. The semantic checker should catch and report all semantic errors in the program. Your error messages need not be identical to those of `coolc`.

A simple error reporting method is supplied

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

This routine takes a `Class_` node and returns an output stream that you can use to write error messages.

8. Testing the Semantic Analyzer

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 analyze. 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.

9. Remarks

The semantic analysis phase is by far the largest component of the compiler so far. Implementing a complete working semantic analyzer may take some time. For your convenience and to fit the time constraint, you can safely assume the following:

- No inheritance. You'll be tested against one or more classes, but no inheritance relationship between them. No need to handle issues related to inheritance.
- No multiple files. You'll be tested against one file only, containing one or more classes.
- Stanford's parser in this phase reports incorrect line numbers. Don't bother yourself with line numbers.

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 straight-forward.

10. How to Start

- Create a new empty folder, and run the makefile inside it.
- In `semant.cc` file, you can find the method `program_class::semant()`. This is the main function for your code.
- You're provided with the object `program_class::classes` of type `Classes`. The object already contains all the classes gathered from the code by the parser from the previous phase. You'll need to iterate on the classes one by one and check the correctness of each class.
- To check the correctness of a class, you're provided with the object `features` of type `Features` inside every class, containing the list of features this class owns.
- To check the correctness of a feature, you need to be aware of the semantic rules of Cool's features. Check the Cool Reference Manual. Every section in the manual discusses a part of the language and discusses the semantics of this part.
- If you found a semantic error and want to report it, use the method `ClassTable::semant_error(Class_)`, it returns an object of type `ostream&`, so you can use it as the following:
  `semant_error(some_class) << "Semantic error found !!" << endl;`

Of course you need the error message to be more informative. This is just an example.

- To check the existence of some class, method, or attribute, you need to use a symbol table as discussed in lab. You're provided with an implementation of a symbol table. Check the `symtab_example.cc` file to know how to use it.
- The output of this phase is either a list of printed semantic errors, or an annotated AST. To annotate the AST, you need to set the `type` attribute in every `Expression` node. Every expression is provided with the following method to set its type
  `Expression set_type(Symbol s)`
  The above method is located in `cool-tree.handcode.h`.
- To check how the annotated AST should look like, or to know the difference between the AST generated by the parser, and the annotated AST generated by the semantic analyzer, you can check the output of the semantic analyzer implemented in the code generation phase. Do the following:
  - Create a new folder and run the following command
    `make -f /usr/class/cs143/assignments/PA5/Makefile`
  - The files of the next phase is generated. One of them is `semant` file.
  - Write some Cool code in a file. Name it "`mytest.cl`".
  - To check the parser's output, run the command
    `./lexer mytest.cl | ./parser mytest.cl`
    To check the semantic analyzer's output, run the command
    `./lexer mytest.cl | ./parser mytest.cl | ./semant mytest.cl`
  - Note the difference between the two printed trees. You can use the above command to test your semantic analyzer against Stanford's. You can print different error messages from Stanford's analyzer, but the number and type of errors should be the same. The AST should be the same as well.

11. Submission

Submissions and discussions are in lab time. No online submissions required. The assignment is published on 2 May week labs, and will be discussed on 9 May week labs.

Good Luck,