CS-GY 6413/CS-UY 3943 — Programming Assignment 4

Jeff Epstein

Introduction

We will write a compiler for the Cool language. Each of several programming assignments will cover one component of the compiler: scanner, parser, semantic analyzer, and code generator. When combined, these components will result in a complete compiler. This assignment covers the semantic analyzer.

Before starting, please make sure that you're using a suitable programming environment, as discussed in previous assignments.

You should complete the assignment in the C++ language, using the framework provided to you. The framework is documented in *A Tour of the Cool Support Code*, available on Classes.

This assignment assumes that you are already familiar with the Cool language. Please refer to the Cool manual, available on Classes, for details on the language.

Collaboration policy Undergraduate students may work on this assignment either alone, or in a group of exactly two people. Both members of a group must be enrolled in the same section: that is, undergrads may not collaborate with grad students. If working in a group, make sure to submit it on Gradescope using the group submission option. Collaboration with anyone not in your group is prohibited. Graduate students must work alone.

Assignment

Your task is to implement a static semantic analyzer for the Cool language.

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 analyzer should reject erroneous Cool 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 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 *A Tour of the Cool Support Code*.

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.

Tree traversal

In the previous assignment, your parser builds an abstract syntax tree. A common approach for dealing with ASTs is a *recursive traversal*, wherein each node of the AST will recursively apply some operation across all its children, starting with the AST's root. For an example of this approach, see the method `dump_with_types`, defined in `dumptype.cc`, which illustrates how to traverse the AST recursively and gather information from it. Most AST classes contain an implementation of this method. Start with `program_class::dump_with_types`, which is the implementation of the method for the AST root.

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`, `cool-tree.handcode.h`, and/or `semant.cc`.

Inheritance

Your semantic analyzer is responsible for checking and enforcing Cool rules about inheritance.

Inheritance relationships specify a directed graph of class dependencies: each node represents a class, and contains a reference to the node of each class that inherits from it. 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 way to do this is to construct a representation of the type graph and then check for cycles.

Cool has restrictions on inheriting from the basic classes `Int`, `String`, and `Bool`. See the Cool manual for details.

It is an error if class `A` inherits from class `B` but class `B` is not defined.

It is an error to redefine any class.

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 passes. 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.

Naming and scoping

A major portion of the semantic analyzer 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, consider this correct program:

```
class Main {
    i : String;
    main() : Object {
        let i : Bool in
        let i : Int in
        i + i
    };
};
```

Here, `i` is declared as a class attribute, and also in two let expressions, one nested within the other. 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. In this case, the expression `i + i` refers to the `i` of type `Int`.

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:

- class attribute definition
- formal parameters of a method
- `let` expression
- branches of a `case` statements

In addition to object names, there are also method names (introduced by defining a method) and class names (introduced by defining a class). It is an error to use any name that has no matching declaration.

Note that the order of declarations need not correspond to the order of use. For example, the following Cool program is correct:

```
class Main {
  main() : Object {
    a()
  };
  a() : Object {
    self
  };
};
```

The `main` function is allowed to refer to methods declared after itself. This means that your semantic analyzer will probably to have to do one pass to collect all names into symbol tables, then do another pass to check that those names are used correctly.

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 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 emitted (using the provided function), with a description of what went wrong. It is easy to give informative error messages in the semantic analysis phase, because it is often obvious what the error is. You are expected 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.

Code generator interface

For the semantic analyzer to work correctly with the rest of the compiler, some care must be taken to adhere to the interface with the code generator. The code generator must know the type of every expression in a Cool program. It is the job of the semantic analyzer to augment the AST with this information.

For every expression node in the AST, you must set its `type` field 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.

Expected output

For correct Cool program, the semantic analyzer will output an augmented AST, as described above. For incorrect programs, semantic analyzer will output 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. Your error messages need not be identical to those of the reference implementation.

To report an error, you should call `ClassTable::semant_error`. This method takes a `Class_` node and returns an output stream that you can use to write error messages. For example:

```
semant_error(myclass) << "Redefinition of basic class " << name << "." <<
    ↪ endl;
```

Since the parser ensures that `Class_` nodes store the file in which the class was defined, 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 evaluated on whether your semantic analyzer correctly annotates ASTs with types and on whether your semantic analyzer works correctly with the reference code generator.

Other notes

The semantic analysis phase is 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?
- What do I need in order to check a requirement?
- How can I collect the information necessary to check a requirement?

If you can answer these questions for each aspect of Cool, implementing a solution should be straight-forward.

Setup

Download two files: `pa4.zip`, which contains starter code specifically for this assignment; and `refimpl.zip`, which contains a reference implementation of the Cool compiler. Extract the content of both zip files into separate directories.

There are a few files that are of interest of us:

- `cool-tree.h` and `cool-tree.handcode.h` — These files contain the structure of the AST. You will probably want to modify them to add additional functions and variables to the relevant classes. However, do not modify the functions and variables already defined.
- `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`, which is declared in 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 may add any additional declarations.
- `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 responsible only 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.

Do not modify any other file.

Running the `make` command from that directory will compile your code. If compilation is successful, it will produce an executable file `semant`.

Testing

There are several ways you can test your parser. In all cases, it's up to you to design inputs that adequately test all features of your parser: don't rely only on the provided test files.

Direct testing You need a working scanner and parser to test the semantic analyzer. You can use either your scanner and parser from the previous assignments, or the scanner and parser from the reference implementation, or some combination. The following `bash` command will call the reference compiler's `lexer` executable, giving it the provided file `good.cl` as input, and piping the resulting token stream as input to the reference compiler's parser and semantic analyzer:

```
../refimpl/lexer good.cl | ../refimpl/parser | ../refimpl/semant
```

This produces the following annotated AST:

```
#1
_program
  #1
  _class
    C
    Object
    "good.cl"
    (
    #2
    _attr
      a
      Int
      #0
      _no_expr
      : _no_type
    #3
```

```
  _attr
    b
    Bool
    #0
    _no_expr
    : _no_type
  #4
  _method
    init
    #4
    _formal
      x
      Int
    #4
    _formal
      y
      Bool
    C
    #5
    _block
      #6
      _assign
        a
        #6
        _object
          x
        : Int
      : Int
      #7
      _assign
        b
        #7
        _object
          y
        : Bool
      : Bool
      #8
      _object
        self
      : SELF_TYPE
    : SELF_TYPE
  )
#13
_class
  Main
  Object
  "good.cl"
  (
  #14
  _method
    main
```

```
      C
      #15
      _dispatch
        #15
        _new
          C
        : C
        init
        (
        #15
        _int
          1
        : Int
        #15
        _bool
          1
        : Bool
        )
      : C
  )
```

The parser phase emits a similar AST, but the output of the semantic analyzer shows the additional type annotations: each expression is augmented with its static type. For example, the assignment statement `a <- x` on line 6 of `good.cl` is represented by the following subtree, which shows that both the expression's right-hand side ("`x`") and the expression overall have type `Int`.

```
        #6
        _assign
          a
          #6
          _object
            x
          : Int
        : Int
```

You can compare the result with your own semantic analyzer by substituting your compiled `semant` executable into the above command line, like this:

```
../refimpl/lexer good.cl | ../refimpl/parser | ./semant
```

It's important that your semantic analyzer correctly handle incorrect Cool programs. The provided `bad.cl` contains examples of some errors you should be able to recognize and report. As you can see below, when errors are found, the semantic analyzer does not emit an annotated AST, but rather prints out the errors and aborts compilation.

```
bad.cl:16: In call of method init, type Int of parameter y does not conform
   ↪ to declared type Bool.
bad.cl:17: Method init called with wrong number of arguments.
bad.cl:18: Dispatch to undefined method iinit.
Compilation halted due to static semantic errors.
```

The AST input and output of your semantic analyzer is handled by the provided AST classes. Your program simply analyzes and annotates the AST. For reporting errors, use the provided `ClassTable::semant_error` mechanism; do not output to `cout` directly.

Testing as part of compiler The provided reference implementation of Cool contains complete versions of all four phases of the compiler. You can replace the reference version of the semantic analyzer with your version, and then verify that the compiler still works correctly.

To replace the semantic analyzer, just copy your compiled `semant` into the directory where you've extracted the reference implementation:

```
cp semant ../refimpl/semant
```

Now run the compiler `coolc` on your test program. If your semantic analyzer is correct, the subsequent phases of compilation should run normally, and you can run the compiled test program.

Rules

Your code should compile cleanly with a 5.x or newer version of `gcc` and the provided `Makefile`.

You may use any standard C++ header files, but you may not use any third-party libraries. You must use the provided AST library.

Do not add any additional files to the program. Modify only those files indicated in `README`.

Use good coding style, including informative variable names, consistent indentation, and clear structure. Your code should be robust and generally free of bugs. As a special case, please note that heap leaks are acceptable, as long as they don't impact the functioning of the program.

You are not obligated to use comments. However, confusing, unclear, and uncommented code may result in a reduced grade. The grader must be able to understand your code, and comments are one way to improve the readability of your code.

Submission

Test your code thoroughly before you submit.

In addition to your code, you must answer the questions at the end of the provided `README` file. Write your answers directly in the `README` file.

Run the command `make zip` and check the content of the resulting zip file to make sure that it contains all the code you wrote. Submit only the zip file on Gradescope. Do not add additional files to the zip file.

Acknowledgments

This sequence of assignments is based on work by Alex Aiken and Westley Weimer.