CS131 Compilers: Programming Assignment 3
Due Thursday, Dec 15, 2019 at 11:59pm

Fu Song Qi Qin

1 Policy on plagiarism

For project assignment 3 and 4, you may work individually or as a pair for those two assignments. While you may discuss the ideas and algorithms or share the test cases with others, at no time may you read, possess, or submit the solution code of anyone else (including people outside this course), submit anyone else's solution code, or allow anyone else to read or possess your source code. We will detect plagiarism using automated tools and will prosecute all violations to the fullest extent of the university regulations, including failing this course, academic probation, and expulsion from the university.

2 Overview of the Project

Project assignments 1-4 will direct you to design and build a compiler for the Classroom Object-Oriented Language (cool), designed by Alexander Aiken for use in an undergraduate compiler course project. Assignments will cover the frond-end of the compiler: lexical analysis, parsing, semantic analysis, and intermediate code generation. Each assignment should be solved using C++ programming language and will ultimately result in a working compiler phase which can interface with other phases.

In this assignment, you will implement the static semantics analysis 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.

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:

- Look at all classes and build an inheritance graph.
- Check that the graph is well-formed.
- For each class:
    1. Traverse the AST, gathering all visible declarations in a symbol table.
    2. Check each expression for type correctness.
    3. 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. This time, you may work individually or **as a pair**.

3 Files and Directories

To get started, you first should download PA3.zip from the class web page. 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.) This is a list of the files that you may want to modify.

1. 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.

2. 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.

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

4. good.cl and bad.cl. These files test a few semantic features. You should add more test files to ensure that you test both several erroneous programs and several correct programs.

5. apt.txt and compiler.json. Refer to the Readme.md for further detail.

6. Makefile(Optional): this file describes how to generate the binaries for parsing. You maybe want to modify it. If you would like to understand Makefile, read `https://www.gnu.org/software/make/manual/make.html`.

It is important to make good tests to ensure that your semantic analyzer is working properly.

Although these files are incomplete as given, the parser does compile and run. To build the parser, you must type

$$make\ semant$$

in the directory PA3/src. This will start the compilation process and link the support code needed for this phase into your working directory. Start the semantic analyser by typing

$$lexer\ input\_file \mid parser \mid semant$$

4 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. 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 (C++) directly. The method implementations you wish to add should go into semant.cc.

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.

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.

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:

1. attribute definitions;
2. formal parameters of methods;
3. let expressions;
4. 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.

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 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 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 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 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.

9 Expected Output

For incorrect programs, the output of semantic analysis is an error message. Assuming the inheritance hierarchy is well-formed, the semantic checker should catch and report at least one semantic error in the program, output some error message with filename and line number.

When you detected a cyclic inheritance structure, you are free to report the errors about the involved classes in any order.

We have supplied you with simple error reporting methods ostream& ClassTable::semant_error(Class_) (C++). This routine takes a Class_ (C++) node and returns an output stream that you can use to write error messages. Since the parser ensures that Class_/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.

10 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 scanner/parser we provided. Even if you use your own scanner and/or parser, it is wise to test your semantic analyzer with the provided scanner and parser at least once, because we will grade your semantic analyzer using our scanner and parser.

You will run your semantic analyzer using *semant* by typing

$$lexer\ input\_file \mid parser \mid semant$$

Note that *semant* takes a *-s* flag for debugging the analyzer; using this flag merely causes `semant_debug` (a global variable in *semant.cc* to be set. Adding the actual code to produce useful debugging information is up to you.

Once you are confident that your semantic analyzer is working, try running our code generator *cgen* 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*.

11 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 straight-forward.

12 When, What and How to Hand in

1. When: Make sure that the final version you submit does not have any debug print statements and that AST outputed by your *semant* is correct.

2. What: You have to hand in the whole repo you cloned from Gitlab. Don't modify any part of the support code (except for Makefile)!

3. How: Login at `http://s3l.shanghaitech.edu.cn:8081/users/sign_in`, fork the template repo at `http://s3l.shanghaitech.edu.cn:8081/compiler/PA3`, **change it to a private repo** and follow the guide in readme.md in the template repo. Otherwise, we will not grade your submission. If you work in pairs, you only need to submit one repo with **both** of your emails in compiler.json file. Note that we will score your submission based on your commits, so **both** of you need to commit to the repo you submit if you work in pairs.