Programming Assignment IV
Due Thursday, June 1, 2017 at 11:59pm

1 Introduction

In this assignment, you will implement a code generator for Cool. When successfully completed, you will have a fully functional Cool compiler!

The code generator makes use of the AST constructed in PA2 and semantic analysis performed in PA3. Your code generator should produce MIPS assembly code that faithfully implements *any* correct Cool program. There is no error recovery in code generation—all erroneous Cool programs have been detected by the front-end phases of the compiler.

As with the semantic analysis assignment, this assignment has considerable room for design decisions. Your program is correct if the code it generates works correctly; how you achieve that goal is up to you. We will suggest certain conventions that we believe will make your life easier, but you do not have to take our advice. This assignment is about twice the amount of the code of the previous programming assignment, though they share much of the same infrastructure. **Start early!**

Critical to getting a correct code generator is a thorough understanding of both the expected behavior of Cool constructs and the interface between the runtime system and the generated code. The expected behavior of Cool programs is defined by the operational semantics for Cool given in Section 13 of the *Cool Reference Manual*. Recall that this is only a specification of the meaning of the language constructs—not how to implement them. The interface between the runtime system and the generated code is given in the *Cool Runtime* manual, available on the course website. See that document for a detailed discussion of the requirements of the runtime system on the generated code. Additionally, you will likely need to read the *Spim Manual* in order to understand the format of the code that you need to output. There is a lot of information in the aforementioned documents, and you need to know most of it to write a correct code generator. *Please read thoroughly.*

You may work in a group of one or two people.

2 Files and Directories

To get started, log in to one of the *corn* machines and run the following command:

```
/usr/class/cs143/bin/pa_fetch PA4 <project_directory>
```

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.) See the instructions in the README file.

Here are the most important files for this project:

- cgen.cc
  This file will contain almost all your code for the code generator. The entry point for your code generator is the **program_class::cgen(ostream&)** method, which is called on the root of your AST. Along with the usual constants, we have provided functions for emitting MIPS instructions, a skeleton for coding strings, integers, and booleans, and a skeleton of a class table (**CgenClassTable**). You can use the provided code or replace it with your own inheritance graph from PA3.

- cgen.h
  This file is the header for the code generator. You may add anything you like to this file. It provides classes for implementing the inheritance graph. You may replace or modify them as you wish.

- emit.h
  This file contains various code generation macros used in emitting MIPS instructions among other things. You may modify this file.

- cool-tree.h
  As usual, these files contain the declarations of classes for AST nodes. You can add field or method declarations to the classes in cool-tree.h. The implementation of methods should be added to cgen.cc.

- cgen_supp.cc
  This file contains general support code for the code generator. You will find a number of handy functions here. Add to the file as you see fit, but don't change anything that's already there.

- example.cl
  This file should contain a test program of your own design. You will likely need to add more example programs to test your code generator comprehensively; the submit script will pick up all files that match the regex *.cl in the root directory of your submission. However, please double check the list diplayed by the submit script before confirming. **It is important to ensure that your code generator is thoroughly tested.**

- README
  This file contains detailed instructions for the assignment as well as a number of useful tips. You should edit this file to include your SUNet ID(s), just as you did in the previous programming assignments.

3 Design

At a high-level, your code generator will need to perform the following tasks:

1. Determine and emit code for global constants, such as prototype objects.

2. Determine and emit code for global tables, such as the **class_nameTab**, the **class_objTab**, and the dispatch tables.

3. Determine and emit code for the initialization method of each class.

4. Determine and emit code for each method definition.

There are many possible ways to write the code generator. One reasonable strategy is to perform code generation in two passes. The first pass decides the object layout for each class, particularly the offset at which each attribute is stored in an object. Using this information, the second pass recursively walks each feature and generates stack machine code for each expression.

There are a number of things you must keep in mind while designing your code generator:

- Your code generator must work correctly with the Cool runtime system, which is explained in the *Cool Runtime* document on the course website. Please ignore all references to CoolAid in that document—we are not supplying that tool this year (but consider following the CoolAid-specific runtime organization anyway).

- You should have a clear picture of the runtime semantics of Cool programs. The semantics are described informally in the first part of the *Cool Reference Manual*, and a precise description of how Cool programs should behave is given in Section 13 of the Manual.
- You should understand the MIPS instruction set. An overview of MIPS operations is given in the spim documentation, which is on the class web page.
- You should decide what invariants your generated code will observe and expect (i.e., what registers will be saved, which might be overwritten, etc). You may also find it useful to refer to information on code generation in the lecture notes.

You do *not* need to generate the same code as coolc. coolc includes a very simple register allocator and other small changes that are not required for this assignment. The only requirement is to generate code that runs correctly with the runtime system.

3.1 Runtime Error Checking

The end of the Cool manual lists six errors that will terminate the program. Of these, your generated code should catch the first three—dispatch on void, case on void, and missing branch—and print a suitable error message before aborting. You may allow SPIM to catch division by zero. Catching the last two errors—substring out of range and heap overflow—is the responsibility of the runtime system in `trap.handler`. See Figure 4 of the *Cool Runtime* manual for a listing of functions that display error messages for you.

3.2 Garbage Collection

To receive full credit for this assignment, your code generator must work correctly with the generational garbage collector in the Cool runtime system. The skeleton contains the **code_select_gc** function, which generates code that sets GC options from command line flags. The command-line flags that affect garbage collection are -g, -t, and -T. Garbage collection is disabled by default; the flag -g enables it. When enabled, the garbage collector not only reclaims memory, but also verifies that "−1" separates all objects in the heap, thus checking that the program (or the collector!) has not accidentally overwritten the end of an object. The -t and -T flags are used for additional testing. With -t the collector performs collections very frequently (on every allocation). The garbage collector does not directly use -T; in coolc the -T option causes extra code to be generated that performs more runtime validity checks. You are free to use (or not use) -T for whatever you wish.

For your implementation, the simplest way to start is to not use the collector at all (this is the default). When you decide to use the collector, be sure to carefully review the garbage collection interface described in the *Cool Runtime* manual. Ensuring that your code generator correctly works with the garbage collector in *all* circumstances is not trivial.

4 Testing and Debugging

You will need a working scanner, parser, and semantic analyzer to test your code generator. You may use either your own components or the components from coolc. By default, the coolc components are used. To change that, replace the lexer, parser, and/or semant executable (which are symbolic links in your project directory) with your own scanner/parser/semantic analyzer. Even if you use your own components, it

is wise to test your code generator with the coolc scanner, parser, and semantic analyzer at least once because we will grade your project using coolc's version of the other phases.

You will run your code generator using mycoolc, a shell script that "glues" together the code generator with the rest of the compiler phases. Note that mycoolc takes a -c flag for debugging the code generator; using this flag merely causes the **cgen_debug** global variable to be set. Adding the actual code to produce useful debugging information is up to you. See the project README for details.

When run on file.cl, mycoolc will produce file.s containing MIPS code generated by your code generator (mycoolc will combine your code generator with the reference parser, lexer, and semantic analyzer). The reference compiler, located at /usr/class/cs143/bin/coolc, works the same way.

In order to debug your code, you might want to perform the following (assume you want to debug code generation on file.cl):

- Run the reference lexer, parser, and semantic analyer, redirecting the output to a file:

```
./lexer file.cl | ./parser | ./semant >file.semant
```

- Load your code generator into gdb:

```
gdb ./codegen
```

- When the gdb prompt appears, set breakpoints and then run the code inside gdb as follows:

```
run <file.semant
```

4.1 Spim and XSpim

The executables spim and xspim are simulators for MIPS architecture on which you can run your generated code. The program xspim works like spim in that it lets you run MIPS assembly programs. However, it has many features that allow you to examine the virtual machine's state, including the memory locations, registers, data segment, and code segment of the program. You can also set breakpoints and single step your program. The documentation for spim/xspim is on the course web page.

Please run spim/xspim using the scripts at:

- `/usr/class/cs143/bin/spim`
- `/usr/class/cs143/bin/xspim`

Warning: One thing that makes debugging with spim difficult is that spim is an interpreter for assembly code and not a true assembler. If your code or data definitions refer to undefined labels, the error shows up only if the executing code actually refers to such a label. Moreover, an error is reported only for undefined labels that appear in the code section of your program. If you have constant data definitions that refer to undefined labels, spim won't tell you anything. It will just assume the value 0 for such undefined labels.

5 Final Submission

Make sure to complete the following items before submitting to avoid any penalties.

- ☐ Edit the README file so that the first line includes your SUNet ID in this format:

  ```
  user: <your_sunet_id>
  ```

  If you are working in a group of two, there should be TWO separate user lines—the submission script will fail if both SUNet IDs are contained on the same line. In particular, if two students with SUNet IDs `abcdef` and `bcdefg` work together, their README should mention the SUNet IDs like this:

  ```
  user: abcdef
  user: bcdefg
  ```

  Your SUNet ID is the name you use to log in to the corn machines (not your 8-digit student ID number), and can be queried with the whoami command.

- ☐ Include your test cases that test your code generator. The submission script will not let you submit unless there is at least one test file, example.cl—however, you should ideally submit more test files.

- ☐ Make sure that you have not modified any file that was symlinked into your project directory. If you created any new source files, make sure they are properly built by your Makefile and that they show up in the list of files found by the `make submit` script.

- ☐ Do NOT log out of corn or close the shell till you receive a "Submission completed." message. If you do not receive this message, it is highly unlikely that we received your submission. Please try to re-submit, and post on Piazza or email us if you aren't able to make it work.