CS 331 Compilers **Fall 2017**

Programming Project 4: COOL Code Generation

Prof. Szajda *Due Tuesday, December 5, 11:59:59 pm*

NOTE: There will be no extensions whatsoever given for this project! So, begin it when you get it!

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 static 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 static 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. I will suggest certain conventions that should make your life easier, but you do not have to take my advice. As always, explain and justify your design decisions in the `README` file. 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 System*. See that document for a detailed discussion of the requirements of the runtime system on the generated code. There is a lot of information in this handout and the aforementioned documents, and you need to know most of it to write a correct code generator. Please read thoroughly. As with the previous projects, you must work in pairs.

2 Files and Directories

To get started, create a directory where you want to do the assignment and execute the following command *in that directory*.

```
make -f /usr/class/cs143/assignments/PA5J/Makefile
```

(As usual, notice the "J" in the path name, and that the numbering is different than our project numbering).

Also 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/PA4J`. Do **not** modify these files. Al-

most 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. (You should already be familiar with most of the other files from previous assignments.) See the README file for details about these additional files.

- `CgenClassTable.java` and `CgenNode.java`

  These files provide an implementation of the inheritance graph for the code generator. You will need to complete `CgenClassTable` in order to build your code generator. You can use the provided code or replace it with your own inheritance graph from PA4.

- `StringSymbol.java`, `IntSymbol.java`, and `BoolConst.java`

  These files provide support for Cool constants. You will need to complete the method for generating constant definitions.

- `cool-tree.java`

  This file contains the definitions for the AST nodes. You will need to add code generation routines (`code(PrintStream)`) for Cool expressions in this file. The code generator is invoked by calling method `cgen(PrintStream)` of class `program`. You may add new methods, but do not modify the existing declarations.

- `TreeConstants.java`

  As before, this file defines some useful symbol constants. Feel free to add your own as you see fit.

- `CgenSupport.java`

  This file contains general support code for the code generator. You will find a number of handy functions here including ones for emitting MIPS instructions. 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. Test as many features of the code generator as you can.

- `README` This file will contain the write-up for your assignment. It is critical that you explain design decisions, how your code is structured, and why you believe your 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.

3 Design

Before continuing, you should read *The Cool Runtime System* to familiarize yourself with the requirements the runtime system imposes on your code generator.

In considering your 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 `classnameTab`, the `classobjTab`, 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 System* manual.
- 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 System* 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 a method, `CgenClassTable.codeSelectGc()`, that generate 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 System* 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 `cgen_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.

4.1 Coolaid

The *Cool Runtime System* manual mentions `Coolaid`, which is a tool used to verify some properties of the MIPS assembly code produced by a Cool code generator. In order to do this, `Coolaid` imposes additional restrictions on the assembly code beyond those required by the runtime system. `Coolaid` is not supported by me and is not necessary for the project, so any `Coolaid`-specific restrictions listed in the *Cool Runtime System* manual may be safely ignored. Even without using `Coolaid`, however, you may find these additional restrictions helpful when deciding how to structure your assembly code.

4.2 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.

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

Submission method is as usual.

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

- Include your write-up in `README`.
- Include your test cases that test your code generator in `example.cl`.
- Make sure all your code for the code generator is in `cool-tree.java`, `CgenClassTable.java`, `CgenNode.java`, `CgenSupport.java`, `BoolConst.java`, `IntSymbol.java`, `StringSymbol.java`, `TreeConstants.java`, and additional `.java` files you may have added.