

Programming Assignment V

Due Friday, December 1

IMPORTANT: The late policy for this assignment is slightly different than for previous assignments. As always, you may turn in the assignment any number of times after the deadline, with the corresponding late penalty. However, we may grade whatever you have turned in at any time after the deadline. The reason for this change is that we need to grade the last assignment promptly.

1 Introduction

In this assignment you will implement code generation for Cool. This assignment is the end of the line: when completed, you will have a fully functional Cool compiler.

The code generator makes use of the AST constructed in PA3 and static analysis performed in PA4. 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 much room for design decisions. Your program is correct if it generates correct code; how you achieve that goal is up to you. We will suggest certain conventions that we believe make life easier, but you don't have to take our advice. As always, explain and justify your design decisions in the **README** file. This assignment is comparable in size and difficulty to the previous programming assignment. Start early!

2 Files and Directories

To get the assignment type

```
make -f ~cs164/assignments/PA5/Makefile
```

in a directory named **PA5**. This command copies a number of files to your directory, some of them with read-only permission. As usual, you should not modify files that are read-only. Please read and follow the directions in the **README** file.

The files that you may need to modify are:

- **cgen.cc** This file will contain your code generator. We have provided an implementation of some aspects of code generation; studying this code will help you write the rest of the code generator. It includes a call to code that will build an inheritance graph from the provided AST. You can use the provided code or replace it with your own.
- **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.
- **cgen_supp.cc** This file contains general support code for the code generator. You will find a number of handy functions here. Modify the file as you see fit, except for the functions **emit_string_constant**, **ascii_mode**, and **emit_string_constant**.
- **emit.h** This file contains code generation macros. You may modify this file.

- **cool-tree.h cool-tree.handcode.h cool-tree.cc** As usual, these files contain the definitions of classes for AST nodes. You can add field declarations to the classes in **cool-tree.h** or **cool-tree.handcode.h**, but add the definitions of the methods to **cgen.cc**; **cool-tree.cc** should not be modified.
- **symtab.h** The symbol table manager should be adequate as is for code generation, but you are free to change it.
- **example.cl** This file should contain a test program of your own design. Test as many features of the code generator as you can manage to fit into one file.
- **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.

There will be many other files in your directory containing other pieces of the compiler. These files were described in previous handouts and are listed in the **README** file. The files we will collect and use to grade your assignment are the ones listed above; you should not modify any other files.

3 Designing and Testing the Code Generator

You will need a working lexer, parser, and semantic analyzer to test your code generator. See the **README** file for instructions on how to use either your own components or the components from **coolc**. It is wise to test your code generator with the **coolc** lexer, parser, and semantic analyzer at least once, because we will grade your code generator using **coolc**'s version.

See the **README** file for instructions for compiling and running a compiler with your code generator. For your convenience, a command line debugging flag **-c** is included in the skeleton, which controls the global variable **cgen_debug**.

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. First, your code generator must work correctly with the Cool runtime system, which is explained in the *Cool Tour* handout. Second, you should have a clear picture of the runtime semantics of Cool programs. The semantics are described informally in the first part of the *CoolAid*, and a precise description of how Cool programs should behave is given in Section 13 of the manual. Third, you should understand the MIPS instruction set. An overview of MIPS operations is given in the **spim** documentation, which is in the course reader and on the class Web page. Fourth, 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 and the text.

4 Extra Credit

There are two ways to earn extra credit on this assignment. The first is the ordinary way: find and document a bug in **coolc**. The second way is to implement some optimization in your compiler.

Extra credit will be awarded for projects that, in addition to code generation, perform some significant optimization of the code. The amount of extra credit depends on how well the optimization is written, documented, and demonstrated. Two critical factors are: (1) correctness (the optimizations don't result in incorrect programs) and (2) the percentage speedup your optimized code achieves over `coolc`, as measured in the number of instructions executed on `spim` over a suite of benchmarks of our choosing.

The total extra credit for optimization will not exceed 5% of the total grade for the course. Roughly speaking, the extra credit is worth up to about half of the two large programming assignments. The final curve for the course will be determined *before* including the extra credit. In other words, if you elect not to do an optimization phase, you will not be at a disadvantage in the final grading with respect to those who do.

This extra-credit option is open-ended; you can do as much as you like. We will award credit for results. A project that merely attempts, but does not complete, an optimization phase may receive as little as no extra credit.

There are many possible optimizations to implement; see the text for ideas. Assuming your initial code generator is straightforward (like `coolc`'s), then two directions that may yield significant improvement are (1) improving register usage and (2) eliminating the test for void in dynamic dispatches if it is safe to do so.

WARNING. We have not implemented an optimization phase in `coolc`, so we can't be sure how difficult it will be. For the same reason, we have no skeleton code to give you—you are on your own. If you want to do an optimization phase, you are encouraged to talk it over with one of the course staff first. *Under absolutely no circumstances should you try optimization before your code generator is finished!!*

There is a `-O` flag that controls the global variable `cgen_optimize`. If you do an optimization phase, it should have no effect unless `cgen_optimize` is 1. We will grade your code generator first with optimization off; this will prevent you from losing points due to bugs in your optimizer.

5 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 `cgen.cc` contains a function `code_select_gc` that 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 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 not to 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 Tour*. Ensuring that your code generator correctly works with the garbage collector in *all* circumstances is not trivial.

6 Spim and XSpim

You will find `spim` and `xspim` useful for debugging your generated code. `xspim` works like `spim` in that it lets you run MIPS assembly programs. However, it has many features that allow you to look at the memory, registers, data segment, and code segment of the code. You can also set breakpoints and single step your program. Look at the documentation for `spim/xspim` in the course reader or in the cs164 home 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.