

Phase IV: Code Generation

Due Date: December 10, 2008.

Teamwork: Highly encouraged.

Purpose:

This project is intended to give you experience in writing a code generator as well as bring together the various issues of code generation discussed in the text and in class.

Groupwork:

The same rules for group work as in Phase II apply.

Project Summary:

Your task is to implement a code generator for Cool. This assignment is the end of the line: when completed, you will have a fully functional Cool compiler, and you will have achieved full Compiler Wizardry status!

The code generator makes use of the AST constructed in Phase II and static analysis performed in Phase III. 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 it generates correct code; how you achieve that goal is up to you. We will suggest certain conventions that we believe will make your 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!

Important Project Files:

- `codeGeneration/CgenSupport.java`

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, but don't change anything that's already there.

- `codeGeneration/CgenClassTable.java` and `codeGeneration/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.

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

These files provide support for Cool constants. You should have a look at these files, especially at the methods `codeDef` and `codeRef`.

- `treeNodes/*.java`

These files contain the definitions for the AST nodes. You will need to add code generation routines for Cool expressions in this file. The code generator is invoked by calling method `cgen()` of class **Program**. Do not modify the existing declarations.

- `generalHelpers/TreeConstants.java`

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

Testing your Code Generator: You will need a working scanner, parser, and semantic analyzer to test your code generator. `testEnvironment/CoolCompiler.java` will automatically run with predesigned, but somewhat buggy prior phases. You may modify `CoolCompiler.java` to use your own phases instead (see comments in the code of `CoolCompiler.java`). If you want stable phases I-III, we recommend using `mycoolc` in the `bin/` directory of the repository, a shell script that “glues” together the generator with the rest of compiler phases. Note that you need to call `mycoolc` using Linux from the directory where your Java project is stored in.

Designing your Code Generator:

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-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-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 in the course handout and on the class Web page, and also available in the resources folder of the repository.
- 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 textbook.

With this in mind, one possible organization for your code generator is:

1. compute the inheritance graph
2. assign tags to all classes in depth-first order
3. determine the layout of attributes, temporaries, and dispatch tables for each class
4. generate code for global data: constants, dispatch tables,...
5. generate code for each feature

Your code generator has to select MIPS instructions to emit, do some kind of register allocation (very simple is fine), and layout the memory for the runtime. You need to decide what strategy to use for each of these within the passes your code generator makes over the AST.

Hints on Getting Started:

Before you dive into writing your code generator, we strongly recommend that you write some small Cool programs, compile them with `coolc` (on Linux), and carefully examine the relationship between

the Cool program and the MIPS assembly program. That is particularly helpful when planning how to produce code for method calls, parameter passing, and other control constructs. Also, write your code generator in an incremental manner, generating code for very simple programs, getting them working, and then incrementally adding other more complex constructs to your code generator and corresponding test cases. Note that you do not have to generate the exact same code as that produced by Coolc. It just needs to maintain the semantics of the Cool program being compiled.

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 skeletons contains a function `CgenSupport.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 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-Manual. Ensuring that your code generator correctly works with the garbage collector in *all* circumstances is not trivial.

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 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. Look at the documentation for `spim/xspim` in the course handout or in 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.

Extra Credit:

You may earn extra credit by implementing some optimization in your compiler. Extra credit will be awarded for projects that, in addition to code generation, perform some 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.

This extra-credit option is open-ended; you can do as much as you like. We will award credit for results. For example, 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. 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) specializing the implementation of the basic classes `Int` and `String`.

WARNING. We have not implemented an optimization phase in `coolc`, so 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 **Flags.cgen_optimize**. If you do an optimization phase, it should have no effect unless **cgen_optimize** is on. We will grade your code generator first with optimization off; this will prevent you from losing points due to bugs in your optimizer.

How to Turn In:

Only complete submissions via the svn-repository will be accepted.