CS-GY 6413/CS-UY 3943 — Programming Assignment 5

Jeff Epstein

Introduction

We will write a compiler for the Cool language. Each of several programming assignments will cover one component of the compiler: scanner, parser, semantic analyzer, and code generator. When combined, these components will result in a complete compiler. This assignment covers the code generator. After completing this assignment, you will have a fully-operational Cool compiler.

Before starting, please make sure that you're using a suitable programming environment, as discussed in previous assignments.

You should complete the assignment in the C++ language, using the framework provided to you. The framework is documented in *A Tour of the Cool Support Code*, available on Classes.

This assignment assumes that you are already familiar with the Cool language. Please refer to the Cool manual, available on Classes, for details on the language. In particular, see the section Operational Semantics.

This assignment assumes that you are already familiar with either x86 assembly language or MIPS assembly language.

Collaboration policy Undergraduate students may work on this assignment either alone, or in a group of exactly two people. Both members of a group must be enrolled in the same section: that is, undergrads may not collaborate with grad students. If working in a group, make sure to submit it on Gradescope using the group submission option. Collaboration with anyone not in your group is prohibited. Graduate students must work alone.

Assignment

Your task is to implement a code generator for the Cool language. You may choose to emit either x86 assembly language or MIPS assembly language.

The code generator takes as input an abstract syntax tree that has been augmented with type information, as emitted by the semantic analysis phase. The code generator will emit assembly language implementing any correct Cool program. Note that if the semantic analysis phase succeeded, we know we have a correct program: the code generation phase is not allowed to fail.

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. 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.

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 given in the Cool manual.

The interface between the runtime system and the generated code is given in *A Tour of the Cool Support Code*. 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.

Assembly language

Your program may emit either 32-bit Intel x86 assembly language or 32-bit MIPS assembly language. You may choose one or the other. Indicate your choice in the `README` file.

This assignment assumes a strong familiarity with the assembly language of your choice, so you should use whichever one you are most familiar with. If you're on the fence, I recommend using MIPS, as it is overall simpler.

The support files provided in this assignment, particularly `emit.h`, presume that you'll be outputting MIPS assembly language. If you instead want to output x86 assembly language, you will need to make appropriate changes to these files.

The Cool reference compiler includes two code generators: one for MIPS (called `cgen`), and one for x86 (called `cgen-x86`). When testing your code generator, make sure you are comparing the output against the correct version.

Depending on which code generator you want to use, you can run the reference compiler in one of two ways: the `coolc` script will run the scanner, parser, semantic analyzer, and MIPS code generator; while the `coolc-x86` script will run the scanner, parser, semantic analyzer, and x86 code generator.

There are several different flavors of x86 assembly language syntax, which differ in the names of their opcodes, order of operands, and other syntax conventions. The most common variants are *Intel syntax* (as used by the assemblers `nasm` and `masm`, among others), and *AT&T syntax* (as used by the GNU assembler, `gas`).

Intel syntax (`nasm`)	AT&T syntax (`gas`)
`mov al, 0x05`	`movb $0x05, %al`

The `coolc-x86` script assumes that the code generator will emit x86 assembly language in `nasm`-compatible syntax. If you prefer to emit another flavor of x86 assembly language, you'll have to modify the `coolc-x86` script, as well. Please discuss this with your instructor before undertaking the assignment.

In any case, your emitted assembly language must be compatible with 32-bit architectures. You may not use any 64-bit registers (`rax`, `rbx`, etc...), 64-bit instructions, or other elements specific to 64-bit assembly language.

Design

Before starting to write code, please read *A Tour of the Cool Support Code* to understand the requirements imposed on your code generator by the Cool runtime system.

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, including prototype objects (*someclass*`_protObj`).

2. Determine and emit code for global tables, such as `class_nameTab`, the `class_objTab`, and the method dispatch tables (*someclass*`_dispTab`).

3. Determine and emit code for the initialization method for each class (*someclass*`_init`).

4. Determine and emit code for each method definition (*someclass*`.`*somemethod*).

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 *A Tour of the Cool Support Code*.
- 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 the formal description.
- You should understand the assembly language architecture that you'll be targeting.
- You should decide what invariants your generated code will observe and expect (i.e., what registers will be saved, which might be overwritten, etc).

Your code generator does *not* need to generate identical code as the reference compiler. The only requirement is to generate code that runs correctly with the runtime system.

Runtime code

Your code generator is responsible only for translating the AST (derived from the source code) into assembly language. Additional code, including the implementation of built-in methods, is provided for you in the *runtime library*. For the MIPS version, this code is in a file named `trap.handler`. For the x86 version, this code is in a file named `coolx86.asm`. In both cases, it is worth perusing the runtime library to understand how your code generator should interface with the provided code.

In particular, the runtime library contains the following:

- implementations of built-in methods:
  - `Object.copy`
  - `Object.abort`
  - `Object.type_name`
  - `String.substr`
  - `String.concat`
  - `String.length`
  - `IO.out_string`
  - `IO.in_string`
  - `IO.out_int`
  - `IO.in_int`
- a main program entrypoint that will allocate an instance of the `Main` class and call the programmer-provided `Main.main` method
- garbage collector
- a special function `equality_test`, which implements the logic of the Cool `=` operator
- error-handling functions `_case_abort`, `_case_abort2`, and `_dispatch_abort`, that terminate the program in case of runtime errors

Runtime error checking

The Cool manual lists six runtime errors that will terminate the program:

- dispatch on void
- `case` on void
- missing `case` branch
- division by zero
- substring out of range
- heap overflow

Of these, your generated code is responsible for handling the first three. If your program detects an attempt to dispatch a method on a void value, it should call the provided special function `_dispatch_abort`. If your program detects an attempt to invoke a `case` on a void value, it should call the provided special function `_case_abort`. And if it detects a missing `case` branch, it should call the provided special function `_case_abort2`.

The remaining errors (division by zero, substring out of range, and heap overflow) will be handled by the Cool runtime system or the operating system.

Garbage collection

The Cool runtime library includes a garbage collector that will automatically deallocate inaccessible Cool objects. In order for the garbage collector to work, your generated code must notify the Cool runtime library of newly allocated objects and changes in pointer references between objects.

For this assignment, the garbage collector is disabled. Do not generate code for handling garbage collection.

Expected output

When emitting assembly language, note that your code generator should always emit to the provided `ostream&`, not directly to `cout`.

Your code generator may not emit errors. All errors should have been caught in previous compiler phases. Therefore, your code generator should be able to emit assembly language for any AST that it's given.

Setup

Download two files: `pa5.zip`, which contains starter code specifically for this assignment; and `refimpl.zip`, which contains a reference implementation of the Cool compiler. Extract the content of both zip files into separate directories.

There are a few files that are of interest of us:

- `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&)` function, 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 the previous assignment.

- `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` and `cool-tree.handcode.h` — As in the previous assignment, these files contain the structure of the AST. You will probably want to modify them to add additional functions and variables to the relevant classes. However, do not modify the functions and variables already defined. Put the implementation of any additional functions in `cgen.cc`.
- `cgen_supp.cc` — Additional helpful functions for code generation are found 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. Test as many features of the code generator as you can.
- `README` — This file will contain the write-up for your assignment. You should explain design decisions, how your code is structured, and why you believe that the 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.

Do not modify any other file.

Running the `make` command from that directory will compile your code. If compilation is successful, it will produce an executable file `cgen`. If you're targeting x86 assembly language, you'll have to rename this file to `cgen-x86` in order to test it with the reference compiler.

Testing

There are several ways you can test your code generator. In all cases, it's up to you to design inputs that adequately test all features of your code generator: don't rely only on the provided test files.

Direct testing You need a working scanner, parser, and semantic analyzer to test the code generator. You can use either your own compiler phases from the previous assignments, or the phases from the reference implementation, or some combination. The following `bash` command will use the refereince compiler's lexer, parser, and semantic analyzer, and send the resulting AST as to your code generator, which will output the assembly language code:

```
../refimpl/lexer example.cl | ../refimpl/parser | ../refimpl/semant | ./cgen
```

You should compare the assembly language produced by the reference compiler with that produced by your code generator. Please note that they don't have to be the same: there are many different ways to correctly produce code for a given Cool program. However, you may find the reference compiler's approach instructive.

Testing as part of compiler The provided reference implementation of Cool contains complete versions of all four phases of the compiler. You can replace the reference version of the code generator with your version, and then verify that the compiler still works correctly.

To replace the code generator, just copy your compiled `cgen` into the directory where you've extracted the reference implementation:

```
cp cgen ../refimpl/cgen
```

Now run the compiler `coolc` on your test program. If your code generator is correct, you can run the produced program.

Runtime debugging When you've run your code generator, you may find that it is generating incorrect assembly language code. You have a few options for debugging the generated code:

We use the `spim` emulator to run assembly language programs. It provides a variety of debugging tools. Please consult the `spim` manual, available on Classes, for more information. In particular, the `xspim` debugger will allow you to graphically step through MIPS assembly language code.

For generated x86 code, you can use any debugger of your choice. The `gdb` debugger is widely available, but can be unwieldy for beginners, so you may want to use `ddd` graphical front-end. Tools such as Ghidra, IDA Pro, or Binary Ninja are probably overkill for this project.

Rules

Your code should compile cleanly with a 5.x or newer version of `gcc` and the provided `Makefile`.

You may use any standard C++ header files, but you may not use any third-party libraries. You must use the provided AST library.

Do not add any additional files to the program. Modify only those files indicated in `README`.

Use good coding style, including informative variable names, consistent indentation, and clear structure. Your code should be robust and generally free of bugs. As a special case, please note that heap leaks are acceptable, as long as they don't impact the functioning of the program.

You are not obligated to use comments. However, confusing, unclear, and uncommented code may result in a reduced grade. The grader must be able to understand your code, and comments are one way to improve the readability of your code.

Submission

Test your code thoroughly before you submit.

In addition to your code, you must answer the questions at the end of the provided `README` file. Write your answers directly in the `README` file.

Run the command `make zip` and check the content of the resulting zip file to make sure that it contains all the code you wrote. Submit only the zip file on Gradescope. Do not add additional files to the zip file.

Acknowledgments

This sequence of assignments is based on work by Alex Aiken and Westley Weimer.