CS131 Compilers: Programming Assignment 4
Due Saturday, January 4, 2020 at 11:59pm

Fu Song Qi Qin

1 Policy on plagiarism

For project assignment 3 and 4, you may work individually or as a pair for those two assignments. While you may discuss the ideas and algorithms or share the test cases with others, at no time may you read, possess, or submit the solution code of anyone else (including people outside this course), submit anyone else's solution code, or allow anyone else to read or possess your source code. We will detect plagiarism using automated tools and will prosecute all violations to the fullest extent of the university regulations, including failing this course, academic probation, and expulsion from the university.

2 Overview of the Project

Project assignments 1-4 will direct you to design and build a compiler for the Classroom Object-Oriented Language (cool), designed by Alexander Aiken for use in an undergraduate compiler course project. Assignments will cover the frond-end of the compiler: lexical analysis, parsing, semantic analysis, and code generation. Each assignment should be solved using C++ programming language and will ultimately result in a working compiler phase which can interface with other phases.

In this assignment, you will implement a code generator *cgen* for Cool which makes use of AST constructed in PA2 and static analysis performed in PA3. When successfully completed, you will have a fully functional Cool compiler! 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 PA3, 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 as early as possible!**

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.*
Remark: Ignore the garbage collection requirement of Cool. You dont have to implement it. Just insert calls to the function *Object.copy* to allocate heap objects whenever needed, and never free these objects.

3 Files and Directories

To get started, you first should download PA4.zip from the class web page. This is a list of the files that you may want to modify.

- *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 PA4.

- *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. Test as many features of the code generator as you can.

- compiler.json. Refer to the Readme.md for further detail.

- Makefile(Optional): this file describes how to generate the binaries for parsing. You maybe want to modify it. If you would like to understand Makefile, read `https://www.gnu.org/software/make/manual/make.html`.

It is important to make good tests to ensure that your code generator is working properly.

Although these files are incomplete as given, the code generator does compile and run. To build the parser, you must type

$$make\ cgen$$

in the directory PA4/src. This will start the compilation process and link the support code needed for this phase into your working directory. Start the code generator by typing

$$lexer\ input\_file \mid parser \mid semant \mid cgen$$

4 Design

Before continuing, we suggest you read *The Cool Runtime System* to familiarize yourself with the requirements on your code generator imposed by the 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, 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 initialization method for 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.
- 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 reference program *cgen*. The only requirement is to generate code that runs correctly with the runtime system.

4.1 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.s*. See Figure 4 of the *Cool Runtime System* manual for a listing of functions that display error messages for you.

5 Testing and Debugging

You will need a working scanner, parser, and semantic analyzer to test your code generator. You may use either your own programs or the binaries provided by us. Even if you use your own components, it is wise to test your code generator with the provided scanner, parser, and semantic analyzer at least once because we will grade your project using ours.

You will run your code generator using *cgen* as above.

5.1 Spim

The executables *spim* is a simulator for MIPS architecture on which you can run your generated code. Spim can be obtained from `http://spimsimulator.sourceforge.net`. It provides several versions of MIPS simulator: spim, PCSpim, Qtspim and xspim.

1. PCSpim. Microsoft Windows front end for Spim.
2. QtSpim. Preferred front end for Spim that works on Microsoft Windows, Linux, and Mac OS X.
3. spim. Text-only (terminal or console) front end for Spim. Works on all platforms.
4. xspim. X-Windows front end for Spim. Works on Linux and Mac OS X. (Built on a very old X Windows library that is no longer supported by the Mac.)

QtSpim is the version of Spim that currently being actively maintained. The other versions are still available, but are no longer maintained or updated. You should use the Cool runtime system *trap.s* as the exception handler for Spim.

Note: I tested QtSpim, spim and xspim on Ubuntu. The generated code by the reference code generator cgen works on spim, but **don't** work on QtSpim and xspim. You might try them again.

For instance, use

$$lexer\ example.cl \mid parser \mid semant \mid cgen > example.s$$

to generate MIPS assembly program of *example.cl*. invoke spim to execute the assembly program by typing:

$$spim\ -exception\_file\ trap.s\ -file\ example.s$$

Details of spim refer to the manual of spim in Documentation of Spim source code.

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.

6 When, What and How to Hand in

1. When: Make sure that the final version you submit does not have any debug print statements and that code generated by your *cgen* can run correctly on spim.

2. What: You have to hand in the whole repo you cloned from Gitlab. Don't modify any part of the support code (except for Makefile)!

3. How: Login at `http://s3l.shanghaitech.edu.cn:8081/users/sign_in`, fork the template repo at `http://s3l.shanghaitech.edu.cn:8081/compiler/PA4`, **change it to a private repo** and follow the guide in readme.md in the template repo. Otherwise, we will not grade your submission. If you work in pairs, you only need to submit one repo with **both** of your emails in compiler.json file. Note that we will score your submission based on your commits, so **both** of you need to commit to the repo you submit.