CS164 Programming Language and Compilers	**Spring 2016**

Programming Assignment V

Assigned: Mar. 28 **Due:** May 9, at 11:59 PM (Checkpoint due Apr 15, at 11:59 PM)

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 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 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. **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 are defined by the operational semantics for Cool given in Section 13 of the *Cool Reference Manual* (docs/cool-manual.pdf from the zip archive, see below). 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* (docs/cool-runtime.pdf from the zip archive, see below) 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.*

You may work in a group of one or two people. The submit program will ask you to specify group members when you turn in your assignment.

2 Files and Directories

Everything you need is included in the archive file (PA5.zip) available from the course website. This package contains a number of files. You should not modify the following files: ASTConstants.java, ASTLexer.java, ASTParser.java, AbstractSymbol.java, AbstractTable.java, Cgen.java, ClassTable.java , Flags.java, IdSymbol.java, IdTable.java, IntTable.java, ListNode.java, build.xml, StringTable.java, SymbolTable.java, SymtabExample.java, TokenConstants.java, TreeNode.java, and Utilities.java. Almost all of these files have been described in previous assignments. In fact, if you modify these files, you may find it impossible to complete the assignment.

This is a list of the files (and folder) that you may want to modify. Most of the other files, you are probably familiar with from the previous assignments.

- 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. These skeletons are much larger than the ones for previous assignments. The skeletons provides three components of the code generator:

  - functions to build the inheritance graph; (we supply this in case you didn't get this working for PA4)
  - functions to emit global data and constants;

  You should work to understand this code, and it will help you write the rest of the code generator.

- 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. You should write a correct Cool program which tests as many aspects of the code generator as possible. It should pass your code generator, and running MARS on the generated output should run the program correctly.

- 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.

- tests
  is a directory containing seven test cases.

3 Instructions

To compile your compiler and code generator, cd into your PA5 directory and type:

“ant cgen”

To test your compiler, type:

“python mycoolc.py [-o output_filename.s] <file1.cl> <file2.cl> ...”

This command parses all the cool files given on the command line, passes them through the semantic checker, and then hands the program AST to your code generator.

To run the reference compiler, type:

“python coolc.py [-o output_filename.s] <file1.cl> <file2.cl> ...”

To run your compiler on the file example.cl and run the resulting MIPS assembly (with output in example.output), type:

“ant test”

To run your compiler on the test files in tests directory, type:

“ant test-all”

To run the produced code:

“python runmips.py <file1.s>” (or the output filename you chose)

To compare your compiler with the reference compiler, type:

“python compare-cgen.py <file1.cl>”

To turn in your work at checkpoint, copy the folder containing your PA5 code to the instructional machine and type there:

“ant submit-clean”
“submit PA5-checkpoint”

To turn in your work finally, copy the folder containing your PA5 code to the instructional machine and type there:

“ant submit-clean”
“submit PA5”

Be sure to submit all relevant source files. In particular, you probably want to turn in cool-tree.java, TreeConstants.java, BoolConst.java, IntSymbol.java, StringSymbol.java, CgenNode.java, CgenClassTable.java, CgenSupport.java, example.cl, README.

You may turn in the assignment as many times as you like. However, only the last version will be retained for grading.

3.1 Using your own machine

If you want to use your own machine for the development, you will need to install JDK 7, Apache Ant 1.7 (or newer), and Python 2.7. You can get the packages from the following links:

- JDK 7: `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`
- Apache Ant: `http://ant.apache.org/bindownload.cgi`
- Python 2.7.11: `https://www.python.org/downloads/`

After installing a package, please put the path to the bin folder of the installed package in your `PATH` variable. In other words, `java`, `javac`, `ant`, and `python` commands should run on the shell prompt without the exact path to the corresponding bin folder. You also need to set `JAVA_HOME` and `ANT_HOME` environment variables.

Mac and Linux. You can also use a package manager for your system, such as `apt-get` and `brew`, to install a necessary package. Just make sure that you are using the correct version. Package managers do not set `JAVA_HOME` and `ANT_HOME` environment variables for you. So, please make sure to set these variables.

Windows. Please check the following guides to setup a development environment on a Windows machine:

- *JDK installation*: `http://docs.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html`
- *Setting the JAVA_HOME Variable*: `https://confluence.atlassian.com/doc/setting-the-java_home-variable-in-windows-8895.html`
- *Python on Windows*: `https://docs.python.org/2/using/windows.html`
- *Installing Apache Ant*: `http://ant.apache.org/manual/install.html`

4 Design

Before continuing, we suggest you read *The Cool Runtime System* (docs/cool-runtime.pdf from the zip archive) 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* (docs/cool-manual.pdf from the zip archive), 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 MARS documentation (`http://courses.missouristate.edu/KenVollmar/MARS/Help/MarsHelpIntro.html`).
- 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 portions of the text, primarily ASU (The Dragon Book) Chapter 9.

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.

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

4.2 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 skeletons contain function **CgenClassTable.codeSelectGc** that generate code that sets GC options from command line flags.

The command-line flags that affect garbage collection are -g 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 flags are used for additional testing. With -t the collector performs collections very frequently (on every allocation).

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 Runtime System* manual. Ensuring that your code generator correctly works with the garbage collector in *all* circumstances is not trivial.

5 Testing and Debugging

You will need a working scanner, parser, and semantic analyzer to test your code generator. By default, the reference implementations are used.

You will run your code generator using mycoolc.py, a python script that "glues" together the generator with the rest of compiler phases. Note that mycoolc.py 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.

5.1 MARS

The executables MARS are simulators for MIPS architecture on which you can run your generated code. MARS has a GUI that lets you run MIPS assembly programs - it can also run them on the command line via the runmips.py script. The GUI 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 MARS is in the course reader or on the course web page.

6 Checkpoint

This is a large project. Midway through the project we have a checkpoint, where we evaluate your progress. At that time, we want your code generation to work correctly on a very small set of programs. In particular, we want your code to work on programs with only one class - Main, which inherits from IO - with one method main which immediately calls out_int with the result of an expression. This expression will not contain any let, new, or case expressions (but may contain other kinds of expressions, including if and while). These restrictions imply that you do not need to implement inheritance or inheritance-based class structures for general classes for the first checkpoint (you can just duplicate the output of the reference compiler for the built-in classes and Main). An example program is in tests/checkpoint.cl. Once your compiler handles these test cases, you should submit the implementation by running "`submit PA5-checkpoint`". The checkpoint submission is worth 40% of your grade for the project and will be evaluated on a variety of test cases.

7 Final Submission

Use "`submit PA5`" to submit your final result. 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.
- ☐ Be sure to answer 'yes' to the submission prompt for files that contain your code.

8 Grading (out of 50 points)

The point breakdown for the PA5 final submission is as follows:

- 38 - for main autograder tests (20 points of which will be assigned during the checkpoint)
- 4 points for the README
  - 4 - thorough discussion of design decisions and choice of test cases; a few paragraphs of coherent English sentences should be fine
  - 2 - vague or hard to understand; omits important details
  - 0 - little to no effort
- 4 points - for example.cl
  - 4 - wide range of test cases added
  - 2 - added some tests, but the scope not sufficiently broad
  - 0 - little to no effort
- 4 points - for code cleanliness
  - 4 - code is mostly clean and well-commented
  - 2 - code is sloppy and/or poorly commented in places
  - 0 - little to no effort to organize and document code