Programming Assignment V[1]
A Code Generator for COOL

PART A: Due November 26, 2014

PART B: Due December 4, 2014

1. Introduction

In this assignment you will implement a code generator for Cool. This assignment is the end of the line: 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 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

Create a subdirectory named MyPA5J (`mkdir MyPA5J`) and change (`cd MyPA5J`) to it. To get the project files, type:

```
make -f ~/cool/assignments/PA5J/Makefile source
```

This will create several symbolic links in the current directory. You should not edit the files pointed to by these links. In fact, if you make and modify private copies of these files, you may find it impossible to complete the assignment. See the instructions in the README file.

- `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.

- `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.

- `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.

[1] This programming assignment is the same one used in the spring 2003 CS 164 course at UC Berkeley. Recycled with permission from Professor George Necula.

- `CoolTree.java`

This file contains 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. 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.

- `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.

At this point you should make sure that you have tagged your current version of the compiler so that any changes made during the code generation phase can be rolled back to this state in case you have to.

Copy to your Eclipse `cool-compiler` project any files in your MyPA5J directory that are not already there. In particular make sure that you do not copy CoolTree.java as this file contains several modifications completed during previous phases.

Open the `Cgen.java` file in your Eclipse project and rename the only reference to the `Program` class to `ProgramAbstract`. Your code should now reflect no compiler errors.

3. Testing the Code Generator

To use your own lexer, scanner and semantic analyzer modules (the front end) you must link to the appropriate classes inside the Cgen.java file. By default, Cgen.java is linked to a default scanner that reads output in AST format from the semantic analyzer. You may want to debug your code generator first with correct output from a working compiler front end. To do this simply leave Cgen.java linked to its default scanner and parser classes and paste AST output into the console when you run the Cgen.java application.

3. Designing the 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 Tour* 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 *CoolAid*, 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 on the class Web page.

- 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 (a.k.a Dragon Book) Chapter 9.

4. 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 reader 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.

5. Details and Hints

The program starts by calling method `cgen()` of class `program` in `CoolTree.java`. Class `CgenClassTable`'s constructor is called by the `cgen()` method of class `program`. The constructor "installs" or creates objects for all the classes, adds them to the `SymbolTable` (`CgenClassTable` extends `SymbolTable`, so it's in the class itself) and builds the inheritance tree. The inheritance tree is stored in a `Vector` named `nds`, which is a member of class `CgenClassTable` and is composed of elements of type `CgenNode`. It then calls the `code()` function which is the one that actually does the real code generation work. It generates code for global data and constants (ints, bools, and strings). Then, the Class Names Table, Objects Table, Dispatch Tables, Object Prototypes, Class Initializations and the Processing of Methods should all be done from here.

The generated code is implemented by a stack machine in which each sub-expression returns its result in the accumulator register ($a0), uses the stack for temporary values and must leave the stack the same as it was before (see the class notes for more details).

The *Class Names Table* includes pointers to all the names of the classes.

For example:

```
class_nameTab:
      .word str_const12      #     Object
      .word str_const13      #     IO
      .word str_const14      #     Int
      .word str_const15      #     Bool
      .word str_const16      #     String
      .word str_const17      #     A
      .word str_const18      #     B
      .word str_const19      #     Main
```

The *Objects Table* contains pointers to the object prototype and init method of each class.

For Example:

```
class_objTab:
      .word Object_protObj
      .word Object_init
      .word IO_protObj
      .word IO_init
      .word Int_protObj
      .word Int_init
      .word Bool_protObj
      .word Bool_init
      .word String_protObj
      .word String_init
      .word A_protObj
      .word A_init
      .word B_protObj
      .word B_init
      .word Main_protObj
      .word Main_init
```

Dispatch Tables contain the pointers to the different functions defined for each class: those inherited from its parent(s) and those defined in the class itself.

For Example:

```
A_dispTab:
      .word Object.abort
      .word Object.type_name
      .word Object.copy
      .word A.foobar
      .word A.helloString
```

Object prototypes are used as an "empty" or "dummy" version of an object of each class.

For Example:

```
B_protObj:
      .word 5
      .word 4
      .word B_dispTab
      .word int_const2
      .word -1
```

`5` is the Class Tag (integer identifying the class of the object), `4` is the object's size (4 words in this case), `B_dispTab` is the pointer to `B`'s dispatch table, `int_const2` represents the default value for `B`'s first and only attribute (which is an `Int`, and thus has an initial value of `0` which is the value at `int_const2`). If `B` had any other attributes they would use subsequent slots as needed. The last slot contains a `-1`; this represents the end of an object and is used by the garbage collector. Other objects created during the program's execution also follow this format. Layouts of subclasses extend their parent class' layout.

A *Class Initialization* gives the code for the `init` method and attributes' initializations used for initializing objects with that class as its type.

The *Processing of Methods* is where the code for each class' methods is generated. Method declarations (including initializations) and method dispatches should follow an Activation Record (AR) convention which is explained in the class notes. Methods for the basic classes are already defined in the runtime system (or the `trap.handler` file).

Arithmetic operations require that a new copy of an `int_const` be created to store the result, because `int_const`'s defined in the program itself must not be overwritten. The creation of new objects is required by the following expressions: plus, sub, mul, divide, neg, "new_", and lets with no initialization. For these cases, you should use the function `Object.copy`, which creates a separate copy of the object pointed to by the accumulator (for example, `B_protObj` or `int_constX`) and stores it in the accumulator.

The `CgenSupport` class contains various useful "*emit*" functions for generating the assembly code for the MIPS instructions.

For example:

```
CgenSupport.emitLoadString(CgenSupport.ACC, symbol, s);
```

Generates the following assembly code:

```
la    $a0 str_const0
```

Where `str_const0` is the string represented by `StringSymbol symbol` (which can be obtained from the string table) and `s` is a `PrintStream`, which in our case is the output file.

When in doubt, create a short and simple COOL program and look at `coolc`'s output MIPS assembly program; but remember that, as stated above, the assembly code generated by your code generator may

not necessarily be the same as that generated by `coolc`.

Even if there is no error recovery or correction, your program must still detect runtime errors (like a dispatch to void) which are explained in the *Cool Manual*, and generate the code necessary for the output MIPS program to abort execution. *Void* is represented by `CgenSupport.ZERO` or the `$zero` register.

6. Work Division

PART A

i. Deliverables

On the first part your program is expected to do the following:

- Set the Int, String and Bool class tags. (0 and 1 are for Object and IO, respectively)
- Emit (generate code for) Class Names Table (class_nameTab)
- Emit Class Objects Table (class_objTab)
- Emit Classes' Dispatch Tables
- Emit Object Prototypes
- Emit Classes' Initializations
- Emit Classes' Methods (assuming they only include the expressions below)
- Emit the code for the following expressions:
  - Integer Constant 'int_const'
  - Boolean Constant 'bool_const'
  - String Constant 'string_const'
  - Assignment 'assign'
  - Addition 'plus'
  - Subtraction 'sub'
  - Multiplication 'mul'
  - Division 'divide'
  - Instantiation 'new_'

Ignore any other expressions that are not in this part. The test cases will not include any expressions that are not of **PART A**.

ii. Turning In PART A

You must perform the following instructions for turning it in:

1. Make sure your code is in `CoolTree.java`, `CGenClassTable.java`, and `TreeConstants.java` and that it compiles and works.
2. Your test program should be in `example.cl`.
3. Include any relevant comments in the `README` file and answer any questions that appear in it.
4. Tag you GitHub project with an appropriate label
5. Submit your GitHub tag by email (before **November 26, 2014 11:59pm**):

PART B

i. Deliverables

On the second part your program is expected to do the following:

- Emit the code for the following expressions:
  - Object Variable 'object'
  - Case 'typ_case'
  - Static Dispatch 'static_dispatch'
  - Dispatch 'dispatch'
  - If-Then-Else-Fi 'cond'
  - While-Loop-Pool 'loop'
  - Block 'block'
  - Let 'let'
  - Negation 'neg'
  - Less than 'lt'
  - Equality 'eq'
  - Less than or equals 'leq'
  - Complement (Not) 'comp'
  - Void Test 'isvoid'
- Generate working MIPS assembly code for any correct Cool program. (i.e. anything else that may be missing)

ii. Turning In PART B

Follow the same steps as **PART A** above except for step 5:

6. Submit your GitHub tag by email (before **December 4, 2014 11:59pm**):