Programming Assignment 3: "Backend"

1 Overview

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 A1 and static analysis performed in A2. Your code generator should produce MIPS assembly code that faithfully implements *any* correct Cool program. There is no error recovery in code generation—all errors that the compiler detects have been detected by the previous phases.

Your program is correct if the code it generates works correctly; how you achieve that goal is up to you. We suggest certain conventions that we believe will make your life easier, but you do not have to take that advice. As always, explain your decisions in the README file.

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 Section A of this handout. See that section 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 you need to know most of it to write a correct code generator. *Please read thoroughly.*

You must work on this assignment in the same team as for A1 and A2.

1.1 Design

At a high-level, the 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.

You do *not* need to generate the same code as coolc, which 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.

There are a number of things you must keep in mind while implementing your code generator:

- Your code generator must work correctly with the Cool runtime system. Please read Section A to familiarize yourself with the requirements on your code generator imposed by the runtime system.
- 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, 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.

1.1.1 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.handler`. See Figure 4 for a listing of functions that display error messages for you.

1.1.2 Garbage Collection

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 Section B.

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.

1.2 Testing and Debugging

You will need a working scanner, parser, and semantic analyzer to test your code generator. You can run your code generator using `mybackend`, a shell script that "glues" together the generator with the rest of compiler components. You may use either your own components or the components from the reference compiler. By default, the reference components are used. Even if you use your own components, it is wise to test your code generator with the `coolc` scanner, parser, and semantic analyzer at least once because we will grade your project using reference compiler components.

You can use the command line flag `-c` for debugging the code generator. Using this flag merely causes a static field `cgen_debug` 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.

The executable `coolspim` is our version of `spim`, a simulator for MIPS architecture on which you can run your generated code. The documentation for `spim` is on 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.

2 Getting and Submitting the Assignment

1. Set up your local git repositories and your team's repository as in the previous assignment. All work on this assignment by all team members should be done and submitted in your team's repository in a branch called `backend` – again, based off the *original* `main` branch, not the `frontend` or `backend` branches.

2. Follow the instructions in the `assignments/pa3/README` file.

3. You may modify all files that match `Cgen*.java`. **You should NOT edit any other files nor add any other files that are required to compile your project.**

4. Again, you will submit a zip archive of your entire `ECS652U-cw-group<number>` code repository, including the (possibly "hidden") `.git` directory.

   We will mark your project based on the latest commit on the `backend` branch (regardless of the commit message) at the deadline.

offset -4	Garbage Collector Tag
offset 0	Class tag
offset 4	Object size (in 32-bit words)
offset 8	Dispatch pointer
offset 12...	Attributes

Figure 1: Object layout.

A The Runtime System

The runtime system consists of a set of hand-coded assembly language functions that are used as subroutines by Cool programs. The use of the runtime system is a concern only for code generation, which must adhere to the interface provided by the runtime system.

The runtime system contains four kinds of subroutines:

1. startup code, which invokes the main method of the main program;
2. the code for methods of predefined classes (`Object`, `IO`, `String`);
3. procedures needed by Cool programs to test objects for equality and handle runtime errors;
4. the garbage collector.

The Cool runtime system is in the file `lib/trap.handler`. This file is loaded automatically whenever coolspim is invoked. Comments in the file explain how the predefined functions are called.

The following sections describe what the Cool runtime system assumes about the generated code, and what the runtime system provides to the generated code. Read Section 13 of the Cool Reference Manual for a formal description of the execution semantics of Cool programs.

A.1 Object Header

The first three 32-bit words of each object are assumed to contain a class tag, the object size, and a pointer for dispatch information. In addition, the garbage collector requires that the word immediately before an object contain -1; this word is not part of the object.

Figure 1 shows the layout of a Cool object; the offsets are given in numbers of bytes. The garbage collection tag is -1. The class tag is a 32-bit integer identifying the class of the object. The runtime system uses the class tag in equality comparisons between objects of the basic classes and in the abort functions to index a table containing the name of each class.

The object size field and garbage collector tag are maintained by the runtime system; only the runtime system should create new objects. However, *prototype objects* (see below) must be coded directly by the code generator in the static data area. The code generator should initialize the object size field and garbage collector tag of prototypes properly. Any statically generated objects must also initialize these fields.

The dispatch pointer is never actually used by the runtime system. Thus, the structure of dispatch information is not fixed. You should design the structure and use of the dispatch information for your code generator. In particular, the dispatch information should be used to invoke the correct method implementation on dynamic dispatches.

For `Int` objects, the only attribute is the 32-bit value of the integer. For Bool objects, the only attribute is the 32-bit value 1 or 0, representing either true or false. The first attribute of String objects is an object pointer to an Int object representing the size of the string. The actual sequence of characters of the string starts at the second attribute (offset 16), terminates with 0, and is then padded with 0's to a word boundary. The value *void* is a null pointer and is represented by the 32-bit value 0. All uninitialized variables (except variables of basic types Int, Bool, and String;) are set to *void* by default.

Scratch registers	`$v0`,`$v1`,`$a0`–`$a2`,`$t0`–`$t4`
Heap pointer	`$gp`
Limit pointer	`$s7`

Figure 2: Usage of registers by the runtime system.

`Main_protObj`	The prototype object of class `Main`	Data
`Main_init`	Code that initializes an object of class `Main`, passed in `$a0`	Code
`Main.main`	The main method for class `Main`, `$a0` contains the initial `Main` object	Code
`Int_protObj`	the prototype object of class `Int`	Data
`Int_init`	code that initializes an object of class `Int` passed in `$a0`	Code
`String_protObj`	the prototype object of class `String`	Data
`String_init`	code initializing an object of class `String` passed in `$a0`	Code
`_int_tag`	a single word containing the class tag for the `Int` class	Data
`_bool_tag`	a single word containing the class tag for the `Bool` class	Data
`_string_tag`	a single word containing the class tag for the `String` class	Data
`class_nameTab`	a table, which at index (class tag) $*$ 4 contains a pointer to a `String` object containing the name of the class associated with the class tag	Data
`bool_const0`	the `Bool` object representing the boolean value false	Data

Figure 3: Fixed labels.

A.2 Prototype Objects

The only way to allocate a new object in the heap is to use the `Object.copy` method. Thus, there must be an object of every class that can be copied. For each class X in the Cool program, the code generator should produce a skeleton X object in the data area; this object is the prototype of class X.

For each prototype object the garbage collection tag, class tag, object size, and dispatch information must be set correctly. For the basic classes `Int`, `Bool`, and `String`, the attributes should be set to the defaults specified in the *Cool Reference Manual*. For the other classes the attributes of the prototypes may be whatever you find convenient for your implementation.

A.3 Stack and Register Conventions

The primitive methods in the runtime system expect arguments in register `$a0` and on the stack. Usually `$a0` contains the `self` object of the dispatch. Additional arguments should be on top of the stack, first argument pushed first (an issue only for `String.substr`, which takes two arguments). Some of the primitive runtime procedures expect arguments in particular registers.

Figure 2 shows which registers are used by the runtime system. The runtime system may modify any of the scratch registers without restoring them (unless otherwise specified for a particular routine). The heap pointer is used to keep track of the next free word on the heap, and the limit pointer is used to keep track of where the heap ends. These two registers should not be modified or used by the generated code—they are maintained entirely in the runtime system. All other registers, apart from `$at`,`$sp`,`$ra`, remain unmodified.

A.4 Labels Expected

The Cool runtime system refers to the fixed labels listed in Figure 3. Each entry describes what the runtime system expects to find at a particular label and where (code/data segment) the label should appear.

`Object.copy`	A procedure returning a fresh copy of the object passed in `$a0`. Result in `$a0`
`Object.abort`	A procedure that prints out the class name of the object in `$a0` Terminates program execution
`Object.type_name`	Returns the name of the class of object passed in `$a0` as a string object Uses the class tag and the table class_nameTab
`IO.out_string`	The value of the string object on top of the stack is printed to the terminal. Does not modify `$a0`.
`IO.out_int`	The integer value of the Int object on top of the stack is printed to the terminal. Does not modify `$a0`.
`IO.in_string`	Reads a string from the terminal and returns the read string object in `$a0`. (The newline that terminates the input is not part of the string)
`IO.in_int`	Reads an integer from the terminal and returns the read int object in `$a0`.
`String.length`	Returns the integer object which is the length of the string object passed in `$a0`. Result in `$a0`.
`String.concat`	Returns a new string, made from concatenating the string object on top of the stack to the string object in `$a0`. Return value in `$a0`
`String.substr`	Returns the substring of the string object passed in `$a0`, from index i with length l. The length is defined by the integer object on top of the stack, and the index by the integer object on the stack below l. Result in `$a0`.
`equality_test`	Tests whether the objects passed in $t1 and $t2 have the same primitive type {Int,String,Bool} and the same value. If they do, the value in `$a0` is returned, otherwise $a1 is returned.
`_dispatch_abort`	Called when a dispatch is attempted on a void object. Prints the line number, from `$t1`, and filename, from `$a0`, at which the dispatch occurred, and aborts.
`_case_abort`	Should be called when a case statement has no match. The class name of the object in `$a0` is printed, and execution halts.
`_case_abort2`	Called when a case is attempted on a void object. Prints the line number, from `$t1`, and filename, from `$a0`, at which the dispatch occurred, and aborts.

Figure 4: Labels defined in the runtime system.

There is no need for code that initializes an object of class `Bool` if the generated code contains definitions of both `Bool` objects in the static data area. The easiest way to ensure these labels are correct in the generated code is to adopt the following naming conventions:

`<class>_init`	for init code of class `<class>`
`<class>.<method>`	for method `<method>` code of class `<class>`
`<class>_protObj`	for the prototype object of class `<class>`

Finally, Figure 4 lists labels defined in the runtime system that are of interest to the generated code.

A.5 Execution Startup

On startup, the following things happen:

1. A fresh copy of the `Main` prototype object is made on the heap and initialized by a call to `Main_init`. The code generator must define `Main_init`. `Main_init` should execute all initialization code of `Main`'s parent classes and finally execute the initializations of attributes in `Main` (if there are any).
2. Control is transferred to `Main.main`, passing a pointer to the newly created `Main` object in register `$a0`. Register `$ra` contains the return address.
3. If control returns from `Main.main`, execution halts with the message "COOL program successfully executed".

B The Garbage Collector

The Cool runtime environment includes two different garbage collectors, a generational garbage collector and a stop-and-copy collector. The generational collector is the one used for programming assignments; the stop-and-copy collector is not currently used. The generational collector automatically scans memory for objects that may still be in use by the program and copies them into a new (and hopefully much smaller) area of memory.

Generated code must contain definitions specifying which of several possible configurations of the garbage collector the runtime system should use. The location `_MemMgr_INITIALIZER` should contain a pointer to an initialization routine for the garbage collector and `_MemMgr_COLLECTOR` should contain a pointer to code for a collector. The options are no collection, generational collection, or stop-and-copy collection; see comments in the trap.handler for the names of the `INITIALIZER` and `COLLECTOR` routines for each case. If the location `_MemMgr_TEST` is non-zero and a garbage collector is enabled, the garbage collector is called on every memory allocation, which is useful for testing that garbage collection and code generation are working properly together.

The collectors assume every even value on the stack that is a valid heap address is a pointer to an object. Thus, a code generator must ensure that even heap addresses on the stack are in fact pointers to objects. Similarly, the collector assumes that any value in an object that is a valid heap address is a pointer to an object (the exceptions are objects of the basic classes, which are handled specially).

The collector updates registers automatically as part of a garbage collection. Which registers are updated is determined by a register mask that can be reset. The mask should have bits set for whichever registers hold heap addresses at the time a garbage collection is invoked; see the file trap.handler for details.

Generated code must notify the collector of every assignment to an attribute. The function `_GenGC_Assign` takes an updated address a in register `$a1` and records information about a that the garbage collector needs. If, for example, the attribute at offset 12 from the `$self` register is updated, then a correct code sequence is

```
sw      $x 12($self)
addiu   $a1 $self 12
jal     _GenGC_Assign
```

Calling `_GenGC_Assign` may cause a garbage collection. Note that if garbage collector is *not* being used it is equally important *not* to call `_GenGC_Assign`.