Machine Problem 3

CS 426 — Compiler Construction
Fall Semester 2019

Handed out: November 5, 2019. Due: November 25, 2019, 5PM

In this assignment, you will complete the intermediate code generation phase of your compiler begun in MP2. You will now add full support for classes and implement all the missing features of COOL, including the builtin IO class and the rest of the run-time library. *Some of the information from the MP2 handout is repeated here for completeness, with changes where necessary. Be sure to read this handout through completely.*

Your code generator should produce LLVM assembly code that faithfully implements *any* correct Cool program (except programs with certain uses of `SELF_TYPE`, as described below), and detects run-time semantic errors. There is no error checking in code generation at compile-time—all erroneous Cool programs that can be detected at compile-time have been detected by the front-end phases of the compiler.

As a simplification, you are not required to support the expression `new SELF_TYPE`, and you are not required to support `SELF_TYPE` as the declared type of an attribute or a `let` variable. We will not use these constructs in our tests. This means that the only uses of `SELF_TYPE` in the Cool program text that you need to support are as the return type of methods. You also *do* need to support the `self` variable and other expressions that have a static type of `SELF_TYPE`.

This assignment gives you some flexibility in how exactly you generate LLVM code for individual Cool constructs. You are responsible for most key design choices, including how to organize the virtual function tables of each class, the individual objects of each class, how to perform dynamic dispatch, and how to implement the basic built-in classes except IO (i.e., `Int, Bool, String`). The I/O routines in class IO are given to you, and an implementation of `case`. Note that there are many key design goals to meet, and there are standard design approaches compilers use to meet these goals. We have discussed these approaches in class or in this handout.

This assignment is conceptually more difficult in terms of COOL implementation than MP2, but should have a much smaller learning curve. Nevertheless, we suggest you get started on it right away.

1 Changes to Code from Phase 1

Your job in this MP is to complete the source code so that when you type `make cgen-2` in directory `mp2/src`, you will build a complete code generator for Cool. Much of the code you write will implement completely new features of Cool that were not addressed in Phase 1 (MP2). However, some of the code involves "turning off" parts of the Phase 1 implementation and replacing it with a different implementation. This section enumerates those changes.

1. `MP3` is enabled when you build `cgen-2`. Look for the hint "ADD CODE HERE" in the true branch of `#ifdef MP3`. The new code you need is described in Section 5, below. Conversely, some of your code from MP2 will now be disabled. This is mainly the code that initiates compilation of `Main::main()`.

2. Function `code_main` now needs two changes:

(a) The LLVM function for method Main::main() is no longer stored in class CgenNode. Instead, this method binding should be looked up where you have stored it, just like any other method-to-LLVM-function binding.

(b) The return value of method Main::main() should be ignored. The call to printf is no longer needed. All exchange of values with the external system is now via the IO class.

3. Finally, you will need to change the handling of primitive values to use boxing/unboxing as appropriate. This is detailed in various places below.

2 Using ValuePrinter

The ValuePrinter class provides some printing methods that support useful operations for MP3. For example, you can use ValuePrinter methods for the following:

- Type definitions for objects and vtables. For example:

```
%Int = type {
        %Int_vtable*,
        i32
}
```

- Constant object declarations. For example:

```
@String.2 = constant %String {
        %String_vtable* @String_vtable_prototype,
        i8* getelementptr ([14 x i8], [14 x i8]* @str.2, i32 0, i32 0)
}
```

3 Cool Runtime

MP3 uses the following files, which implement the Cool runtime and were given to you previously as part of the MP2 release. After code generation, you must link against these files to produce a functioning Cool program (the Makefile is set up to do this).

1. `coolrt.c`:
   This is the source code for the Cool runtime. We have given you C implementations of the four methods of class IO that do actual I/O, plus two methods of class Object that are used by these IO methods. You are responsible for adding:

   - definitions for the vtables of the builtin classes;
   - definitions for default value objects; and
   - C functions for all the missing methods of the builtin classes.

   When you run `make` in the test directory, this file will be compiled to native code by Clang, and linked into the program built from each test file. *In its current state, this file will not compile.* You will first need to define the object and vtable layouts.

Suggestion: To work incrementally, start with Ints and a trivial object layout with no vtable, and add a few methods for class Int. That should allow you to use `IO::out_int` and `IO::in_int` quickly. Then expand the object layout and add the vtable as you add more features to your compiler.

2. `coolrt.h`:
   The header for the Cool runtime. It contains only a few function declarations and skeleton type declarations. You are responsible for adding:

   - Type definitions for the vtables of the builtin classes;
   - Type definitions for the objects of builtin classes; and
   - Function declarations for functions defined in `coolrt.c`.

4 Testing the Code Generator

For convenience, we have created a directory *test-2* that you can use to test your code generator. This directory contains one trivial example to demonstrate the test framework. You may want to use the COOL examples from the course resources page, but they are too large to be useful for initial testing so, once again, *you should write your own test cases to test your compiler.* Use separate simple tests initially, e.g., a single constant and simple arithmetic with two constants, and then work your way up to more complex expressions. A few days before the due date, we'll provide our own test suite, as we did for MP2.

The `test-2` directory contains its own Makefile. Some of the targets it provides are:

- `make file.ast`: compile the Cool program file.cl to an AST.
- `make file.ll`: compile the Cool program AST to LLVM assembly
- `make file.bc`: create an LLVM bytecode file from file.ll
- `make file.exe`: create a linked executable from file.bc, linking in the COOL runtime (`coolrt`).
- `make file.out`: execute file.exe and put the output in file.out.
- `make file.verify`: verify your LLVM code obeys LLVM language rules.
- `make file-opt.bc`: create an optimized LLVM bytecode file from file.bc. This is just so you can see whether your code can be optimized effectively by available techniques in LLVM.

Outside the makefile, there is a shell script `checkref.sh` which will compare `file.out`, if it exists, with `file.refout` (which is assumed to exist), so you can provide and test against expected results.

To be sure that you generate correct LLVM code you should call the LLVM verification path with every program that you generate. You can do this by saying `make` *file*`.verify` as described above. See the target `%.verify` in `mp2/Makefile.common` for the command used.

As with Phase 1, you should generate the LLVM `main()` function explicitly using LLVM IR features. See the information above about how the the function `CgenClassTable::code_main()` needs to change for Phase 2 compared with Phase 1.

Note that your code generation phase executable *cgen-2* takes a *-c* flag to generate debugging information. This is set whenever you define *debug* true in your Makefile (the default). Using this flag merely causes `cgen_debug` (a global variable) to be set. Adding the actual code to produce useful debugging information is up to you. See the project *README* for details.

Make sure that you use the right approach for debugging various kinds of bugs. Use a debugger with breakpoints when you need to inspect internal state. Use printfs to track progress or to identify when a value changes, but use a debugger with watchpoints when you need to monitor values that arenÕt easy or useful to print out with printf.

5 Designing the Code Generator

The following sections describe the complete work of MP3, including features implemented in Phase 1 and Phase 2. Some of it is repeated from MP2 but with boxing and unboxing even the handling of Int is different, so make sure to read this through completely.

A key part of this MP is to think about the major design issues, such as object layouts, object initialization, virtual and static method dispatch, etc., and figure out for yourself how to do this. Sections 6 and 7, below, give you some detailed guidance in how to go about figuring out these issues. Close to the deadline, we will provide a binary for the reference solution, *cgen-2-ref*, so you can see what code our compiler generates, and debug errors in your generated code.

There are many possible ways to write the code generator, even assuming a standard "*bottom-up*" strategy. One reasonable strategy is to perform code generation in two passes; this is the strategy used by our solution and by the skeleton code. The first pass decides the object layout for each class, i.e. which LLVM data types to create for each class, and generates LLVM constants for all constants appearing in the program. Using this information, the second pass recursively walks each feature and generates LLVM code for each expression.

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

- 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 12 of the manual.
- You should have a clear picture of LLVM instructions, types, and declarations.
- Think carefully about how and where objects, let-variables, and temporaries (intermediate values of expressions) are allocated in memory. The next section discusses this issue in some detail.
- You should generate unoptimized LLVM code, using a simple tree-walk similar to the one we discussed in class. Focus on generating reasonably efficient local code for each tree node, e.g., wherever possible, avoid extra casts, use getelementptr to index into objects (i.e., to compute the addresses of a structure field), use appropriate aggregate types, etc.
- Ignore the garbage collection requirement of Cool. You don't have to implement it. Just insert `malloc` instructions to allocate heap objects whenever needed, and never free these objects.

6 Representing Objects and Values in COOL

A major part of your compiler design is to develop the correct representation and memory allocation policies for objects and values in COOL, including explicit variables, heap objects, and temporaries. In MP 3, you need to support all kinds of COOL objects, including primitive values.

Here are the guidelines you should follow:

- All values in Cool are objects, including literals. For primitive values, however, you should box/unbox them to/from Cool objects only when needed. When you are finished with your compiler, the result of every one of your primitive-type expressions should be a virtual register and not an object. If you implemented Int or Bool constants as globals in MP2, you should change that so they are used directly as immediate operands in instructions.
- Think of let-variables as names for locations holding values, i.e., pointers to COOL objects: this is the correct interpretation for COOL (and other imperative languages) because the same variable can be assigned different values (and so must point to different heap objects) at different places within its let-block.

  Since a let-variable has a local scope, we can allocate it in the current stack frame using the alloca instruction.
- A superclass object should appear as a nested struct within a subclass object, and in a specific position that you should think about.
- There should be only a single vtable pointer in each object.
- In your generated code for method dispatch or for accessing data fields, you should try to avoid the LLVM 'bitcast' instruction. It is possible to arrange your object representation so a `bitcast` is only needed when retrieving the vtable pointer from an object.[1]
- You will need to include support for run-time type checks (for `case` in particular). Most of the code is provided and described in the next section, but you will have to accommodate it in your object representation.

7 How to attack this project

Since writing a code generator is a fairly big task, we suggest that you go for the following steps in order to build your compiler. *These steps have been tailored for MP3.* Again, make sure to test each portion of code as you complete it!

1. Think about how to represent a Cool object and the vtable for each class in LLVM. How do you deal with inherited classes and their attributes? You can ignore the run-time support for type checking (`case`) at this point.

[1] Explicit type type conversions in the COOL program obviously need casts. These include *upcasts* (using a subclass object within an expression of superclass type) and *downcasts* (using a superclass object as a subclass, which can only be done using a `case` statement in COOL).

2. Once you have decided on a class layout, modify *coolrt.h* and *coolrt.c* accordingly to implement some of the built-in classes and their methods, e.g., enough to support Int operations including I/O. (Do the rest at any time – we don't spell that out.)

   You can even write simple C programs to call these methods directly and test them.

3. Implement *CgenNode::layout_features()*. This will involve visiting each feature of a class and doing some kind of setup. For example, laying out a method might involve creating the corresponding LLVM function (with correct type and formal parameters but empty body) and assigning it a slot in the vtable for the class. You will also need to record the binding of COOL methods to LLVM functions. Similarly, you will have to assign LLVM types for attributes, and a slot in the object layout.

   Now that the features have been laid out, you can create the LLVM `Type` for each class and for its vtable. Exactly how this is done will depend on how you decided to layout the classes in the runtime. Your code generator needs to match that layout.

   Now you can create the actual vtable for each class. This is the global constant that contains the information and function pointers for the class. At this point, your output code should have:

   - A type for the objects of each COOL class.
   - A type for the vtable of each COOL class.
   - Empty methods for the methods of all COOL classes.
   - The vtable of each COOL class.

4. It is time now to promote string constants into real honest-to-goodness objects. You can use the *code_string_table* and *code_def* methods in *cgen.cc* to create a single definition for each unique constant, all of which conveniently appear in the string tables.

5. Generate code for static dispatch. Remember that a Cool method may return `SELF_TYPE`, which you must handle as a special case. Once you get this far, you can construct constant objects and use the runtime's `IO` methods to get some real output from your generated programs.

6. Next, implement dynamic dispatch. Since you've already created your vtables, this is only a minor change from static dispatch (and the two should share most of the implementation in your compiler).

7. You can now modify some of the expression types that you implemented previously in Phase 1; often, few changes should be needed:

   - Arithmetic expressions. Remember that operations on `Int` and `Bool` expect and return values directly in LLVM virtual registers.
   - `let`. By now, you can compile and run simple programs comparable to Phase 1 (without control flow) but with real primitive objects. Let variables for `Int` and `Bool` should be primitive values on the stack, rather than pointers to the heap.
   - `loop` and `if-then-else`. These are now supposed to be using the typing rules for the full language.
   - Assignment. One key change from Phase 1 is that the value being assigned may have a static type that is different from the LHS variable type. For example, when assigning a `String` to an `Object` variable. This is one of the times where you will have to use a `cast` instruction to keep LLVM happy.

Also, this is the point where you will implement boxing. If an `Int` or `Bool` is being converted to `Object`, you will need to allocate an object record on the heap.

To support the provided code for `case` and for modularity, this conversion should be implemented in the `conform` method in `cgen.cc`

8. Implement code generation for `new` and `init`. Make sure that the new object's attributes are initialized in the correct order and that the correct vtable pointer is stored.

   This little step gets you the ability to compile vastly more COOL programs, in fact, any correct program that does not use `case`!

9. Implement `case`.

   We provide code to support one solution.

   Each class is given an integer tag according to a walk over the inheritance tree. This way all the subclasses of any given class have consecutive tags. Testing whether a class is a descendant of another can be done by testing whether its tag is in the range of tags of the descendants of the other.

   For the code

```
class A {};
class B inherits A {};
class C inherits A {};
class D {};
```

   we might assign the following tags and ranges

Class	Tag	Range
A	1	1-3
B	2	2-2
C	3	3-3
D	4	4-4

   An object of dynamic type B can be recognized as a descendant of A by checking that its tag of 2 is in the range 1-3. An object of dynamic type D can be rejected as a descendant of A by checking that its tag of 4 does not fall in into A's range.

   The supplied `typcase_class::code` and `branch_class::code` methods generate code according to this strategy.

   Half of the information necessary for generating the code is the range for a class. The supplied code finds the ranges in the `tag` and `max_child` fields of `CgenNode`. These fields are set by `CgenClassTable::setup_classes`.

   The other necessary information is the class tag corresponding to the dynamic type of the object being examined by the `case`.

   You must figure out where to store the tags in your object or vtable representation, and implement the method `get_class_tag` which emits code to retrieve it. The `CgenNode` argument is for the *static* type of the reference, so you cannot just return the tag of that node.

10. The final step. Implement runtime error handling, if you haven't already. There are only a few cases you need to check, and they're listed in the back of the Cool manual.

    Now you should thoroughly test your compiler. You can use the Cool files in the *examples* directory, but you should also make your own tests to stress individual cases.

8 What and how to hand in

You have to hand the following files:

- *cgen.cc*, *cgen.h*
- *cool-tree.handcode.h*
- *stringtab.handcode.h*
- *value_printer.cc*, *value_printer.h* (even if unchanged)
- *operand.cc*, *operand.h* (even if unchanged)

The four files, *value_printer.*{*cc,h*} and *operand.*{*cc,h*} should not need to be changed, but some students choose to modify them to support their code generator. Hand them in whether or not you change them.

Don't copy and modify any part of the support code! The provided files are the ones that will be used in the grading process.