

# Machine Problem 3

CS 426 — Compiler Construction  
Fall Semester 2023

Handed out: Oct. 10, 2023; due: <u>Nov. 3, 2023, 11:59 pm</u>
---

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 hrough 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, and how to perform dynamic dispatch. The implementations of built-in classes (`IO`, `String`, `Int`, `Bool`) are given to you in `coolrt.{cc,h}`, as well as an implementation of `case` (included in the file `cgen-case.cc`). 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.

## 1 Changes to Code from MP2

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 MP2. However, some of the code involves “turning off” parts of the MP2 implementation and replacing it with a different implementation. This section enumerates those changes.

1. The C macro `MP3` is enabled when you build `cgen-2`. Look for the hint `// TODO: add code here` under `#ifdef MP3`. The new code you need to write is described in Section 3, 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:

- You no longer directly generate code for `Main::main()` using `codeGenMainmain()`. Instead, `Main::main()` should be just like any other Cool method, and in `code_main` you look it up from where you have stored it.
  - 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 Testing the Code Generator

You can use the directory `test` in your MP2 workspace for testing your MP3 code generator; use `make mp3=true <target>` to let the Makefile know you are testing for MP3. 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 like in MP2.

The Makefile in `test` provides a number of targets, including (also listed in MP2 handout PDF):

- `make <file>.ll`: run your code generator `cgen-2` to compile the Cool program `<file>.cl` to LLVM Bitcode (text format);
- `make <file>.verify`: verify your `<file>.ll` obeys LLVM language rules;
- `make <file>-o3.ll`: optimize `<file>.ll` with `opt -O3`;
- `make <file>.bin`: create a linked executable from `<file>-o3.ll` and `coolrt.o`;
- `make <file>.out`: execute `<file>.bin` and put the output in `<file>.out`;
- `make clean`: delete all generated files.

Remember to add `mp3=true` to all these commands, or you can modify your Makefile locally – change where it says `mp3 = false` into `mp3 = true`.

As with MP2, you need to generate an LLVM `main()` function explicitly as the entry point of your generated program. See Section 1 for how the function `CgenClassTable::code_main()` needs to change in MP3.

Your MP3 code generation executable `cgen-2` also 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.

## 3 Designing the Code Generator

The following sections describe the complete work of MP3, including features implemented in MP2 and MP3. Some points are repeated from MP2 but with small differences – due to boxing and unboxing (Section 4.1), even the handling of `Int` is now 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. Section 4 and Section 5 below give you some detailed guidance in how to go about figuring out these issues.

The code skeleton we provided and our reference solution perform code generation in three passes. 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 and third passes recursively walk each feature and generate the needed allocas and the LLVM code, respectively, for each expression. There is now a major amount of work to do in the first pass, which is new to MP3 (compared to MP2).

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 do not need to implement it for the base version of MP3. Just insert `malloc` instructions to allocate heap objects whenever needed, and never free these objects.

## 4 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 MP3, 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 keep them in “unboxed” representation and only box them when needed (Section 4.1). The result of every 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.
- You should create exactly one vtable *type* (as an LLVM struct) and exactly one vtable *instance* (as an LLVM global value) per class. There should be only a single vtable pointer in each object, pointing to the global vtable instance of the object’s dynamic type. The vtable declarations for Cool built-ins (`String`, `I0`, `Int`, `Bool`) are provided in C in `coolrt.h` for you to check out.

- 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.<sup>1</sup>
- 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.

We have effectively provided you the layout of the built-in `IO`, `String`, `Int`, and `Bool` classes in `coolrt.cc|h`. Look there for examples to help you understand how you should proceed for general Cool classes.

## 4.1 Boxed / Unboxed Objects

Cool `Int` and `Bool` hold hardware-supported primitive types (`i32` and `i1` in LLVM). Arithmetic and logical operations on them (`add`, `sub`, `and`, `or`, etc.) are very frequent in programs and cheap on most hardware. When viewing `Int`s and `Bool`s as Cool `Objects` though, these operations are realized as method calls on objects, bringing the entire overhead of vtable querying and function calls to every integer addition.

To avoid so, we give these types an “unboxed” representation where they are simply stored as the corresponding LLVM primitive type on stack. `Unbox[Int] = i32`, and `Unbox[Bool] = i1`. In your code generator, primitive Cool types should stay unboxed as much as possible. This means

- Cool literals of `Int/Bool` types should be emitted as LLVM constants,
- `Int` arithmetic operations and `Bool` logical operations should be LLVM instructions that operate on and produces `i32/i1` directly.
- Cool types that contain `Int/Bool` should be realized as LLVM struct that contain `i32/i1`.
- After calling a function that returns a boxed `Int/Bool`, unbox the return value on the spot – `load` the primitive value from within the `Int/Bool` struct.

The boxed representation is still useful when `Object` methods are called on them:

```
let x: Int <- 1 in x.type_name()
```

and this is when you need to *box* up the unboxed representation of `Int/Bool`. Briefly put, you need to call `Int`’s member functions `Int_new` and `Int_init` (or `Bool`’s) to allocate a boxed representation.

It’s recommended that you implement boxing/unboxing in the `conform()` function in `cgen.cc`; read the document of that function for more details.

## 5 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

---

<sup>1</sup>Explicit 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).

(`case`) at this point. Look at `coolrt.{cc|h}` for the layout of the built-in classes and make sure your plan for Cool class layout is compatible with them.

2. 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 declaring 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.

Now you can create the actual vtable for each class, which is *a global constant* that contains information and member 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.
3. It is time now to promote string constants into real objects. You can use the `code_string_table` and `code_def` methods in `cgen.cc` to create a single definition for each string in the string table.
  4. 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 `I0` methods to get some real output from your generated programs.
  5. Next, implement dynamic dispatch. Since you've already created your vttables, this is only a minor change from static dispatch, and the two should share most of the implementation in your compiler.
  6. You can now re-inspect the expression types that you implemented previously in MP2; often few changes are needed:
    - Arithmetic expressions: arithmetic expressions on `Int` and `Bool` should produce LLVM operators that directly operate on and return LLVM virtual registers.
    - `let`-expression: `let` variables for `Int` and `Bool` should be primitive values on the stack, rather than pointers to the heap.

At this point, you can compile and run simple programs comparable to MP2 (without control flow) but with real primitive objects.

- `loop` and `if-then-else`. These are now supposed to be using the typing rules for the full language.
- Assignment. One key change from MP2 is that the value being assigned may have a different static type from the LHS variable type. For example, assigning a `String` to an `Object` variable. This is when you may have to use a `cast` instruction to keep LLVM happy. Also, this is 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`.

7. Implement code generation for `new`. 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**!

- Implement **case**; we provide a possible solution for **case** code generation, in `cgen_case.cc` in the hand-out, but you still need to pay attention to some class metadata (also mostly done for you):

Each class is given an integer tag in the initial walk over the inheritance tree. All the subclasses of any given class have consecutive tags. Testing whether a class **A** descends from **B** is just a matter of checking if **A**'s tag is in the range of tags of **B**'s descendants.

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 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 4 is not in **A**'s range 1-3.

`typcase_class::code` and `branch_class::code` provided in `cgen_case.cc` uses this exact strategy.

In the code skeleton provided, the information above is maintained in members `tag` and `max_child` of `CgenNode` instances (recall that each `CgenNode` instance correspond to a Cool class). The `tag` member holds the tag of this class, and the descendant range is `[tag, max_child]` (inclusive on both ends). These fields are set by `CgenClassTable::setup_classes`.

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

- 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.

You should thoroughly test your compiler at every major milestone.

## 6 Using ValuePrinter

The `ValuePrinter` class supports useful LLVM operations; for example, the following usages are likely new to you in MP3:

- Type definitions for objects and vtables. For example:

```
ValuePrinter vp(&stream);
// int_vtable_ptr_ty = ..., llvm_i32_ty = ...
vp.type_define("Int", {int_vtable_ptr_ty, llvm_i32_ty});
```

to produce

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

- Global constant struct value. For example:

```
ValuePrinter vp(&stream);
std::string identifier = get_llvm_name();
global_value str_obj_const(string_type, "String.2");
// str_vtable_ptr_ty = ..., i8_ptr_ty = ...
// str_vtable_init = ..., ptr_to_str_const = ...
vp.init_struct_constant(
    str_obj_const, {str_vtable_ptr_ty, i8_ptr_ty},
    {str_vtable_init, ptr_to_str_const});
to produce
@String.2 = constant %String {
    %_String_vtable* @_String_vtable_prototype,
    i8* getelementptr ([14 x i8], [14 x i8]* @str.2, i32 0, i32 0)
}
```

If your MP2 solution is in LLVM API, you are free to migrate to using `ValuePrinter`, stick to LLVM API, or use a mix of both if possible.

## 7 What and How to Submit

You need to hand the following files:

- `cgen.cc`, `cgen.h`
- `cool_tree.handcode.h`
- `stringtab.handcode.h`
- `coolrt.cc`, `coolrt.h` (even if unchanged)
- `value_printer.cc`, `value_printer.h` (even if unchanged)
- `operand.cc`, `operand.h` (even if unchanged)

The files `coolrt.{cc,h}`, `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.

The hand-in program `/class_cs426/public/cs426_handin` is still available on your virtual machine, and will start to accept mp3 submission a few days after the project's start date. To hand in your solution, run the program with 2 arguments – the name of the MP and the `src/` directory:

```
/class_cs426/public/cs426_handin mp3 <path_to_src_dir>
```

The submission script will run your code generator on a few sampled tests, until we release most of test cases close to the deadline. As noted, we will give you enough time to test and fix any remaining bugs you missed during your own testing, but by the time of the test cases release, it will not be enough time to make major changes to your code generator.

You can hand in the MP multiple times. The one we will grade is your last handin.

**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.

**If you used LLM-based code-generating tool in your submission**, follow the same rules as in MP1:

- If you used a prompted code generator, such as GPT-4, annotate every block of code that was assisted by the code generator (even if you further edited the code generator's output). You should label and number each such code block with comments (e.g., `/* LLM Block 1 */`) and clearly indicate the start/end line of the block. Comment at the end of the file on (1) the tool you used (GPT-3.5, GPT-4, etc.) and (2) the full prompt you gave to the tool, for each code block.
- If you used an unprompted code generator, such as GitHub Copilot, comment at the beginning of each file the name of the tool you used, and indicate any large block of code ( $\geq 5$  lines) you got from the tool.
- We will manually check these comments and may try to reproduce some of the code generation results with your prompts.

## 8 Extra Credit

Implement garbage collection for your Cool programs. You may choose

- a tracing algorithm, such as mark-and-sweep – you decide how frequently to run the mark-and-sweep collection action, or
- a reference counting algorithm.

In both cases, your design decision should be reasonable and well-documented, as we will read your implementation in addition to using automated testing.

We will run your garbage-collected binary on the same test cases as used for the base version of MP3. *Your garbage collector should at least execute once at the exit of `Main::main()`* to deallocate all the heap-allocated memory; this is also what our automated testing checks for. We also encourage you to design your GC so that it runs more frequently (but not too frequently) at reasonable locations, which will be critical in real-world scenarios.

Doing so requires some changes to both your code generator and the Cool runtime. In the code generator, you likely need to emit some runtime function calls around every `new` and assignment site. In the Cool runtime, you then need to provide a definition for these functions and maintain some global states. You can safely assume that no synchronization primitives (mutexes, etc.) are needed as Cool programs are always single-threaded.

To assist your debugging and our automated testing, we provide a dynamic library that shadows GLIBC definitions of `malloc` and `free`. Your *generated program* (not your code generator) should always link against it. Every time your program calls `malloc` and `free`, a line will be printed into `stderr`, with your program's `stdout` uninterrupted. We also provide an updated `test/Makefile`. You can find them as `src_gc/libmp3gc.so` and `src_gc/Makefile` in the handout directory, and you'll need to copy both into your `test` directory.