Machine Problem 2

CS 426 — Compiler Construction
Fall Semester 2017

Handed out: September 21, 2017. Due: October 12, 2017, 5:00 p.m.

In this assignment and the next (MP 2 and MP 3), you will implement the intermediate code generation phase of your compiler. In MP 2, you'll implement only the language features of COOL shown in the figure below.

1 COOL Language Subset for MP 2

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{class}; \\
\textit{class} & ::= & \textbf{class Main}\ \{\ \textit{feature};\} \\
\textit{feature} & ::= & \textbf{main}()\ :\ \textbf{Int}\ \{\ \textit{expr}\ \} \\
\textit{expr} & ::= & \text{ID}\ \texttt{<-}\ \textit{expr} \\
& | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{expr}\ \textbf{else}\ \textit{expr}\ \textbf{fi} \\
& | & \textbf{while}\ \textit{expr}\ \textbf{loop}\ \textit{expr}\ \textbf{pool} \\
& | & \{\ \textit{expr};^{+}\ \} \\
& | & \textbf{let}\ [\![\text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ]]\!],^{+}\ \textbf{in}\ \textit{expr} \\
& | & \textit{expr} + \textit{expr} \\
& | & \textit{expr} - \textit{expr} \\
& | & \textit{expr} * \textit{expr} \\
& | & \textit{expr} / \textit{expr} \\
& | & \tilde{}\,\textit{expr} \\
& | & \textit{expr} < \textit{expr} \\
& | & \textit{expr} <= \textit{expr} \\
& | & \textit{expr} = \textit{expr} \\
& | & \textbf{not}\ \textit{expr} \\
& | & (\textit{expr}) \\
& | & \text{ID} \\
& | & \text{integer} \\
& | & \textbf{true} \\
& | & \textbf{false}
\end{array}
$$

Figure 1: Syntax for the subset of COOL to be implemented in MP 2.

Essentially, you are leaving out language features that require implementing classes, including methods and method calls, attributes, inheritance, constructors, **new** and **case**. The class **Main** is the only class, and it is assumed to have a single method **Main.main() : Int**. This method becomes a simple, global

LLVM function. (We have given you special-purpose code to translate class Main and method main(); you only need to translate its body.)

Note that the only types you must support are `Int` and `Bool`; in particular, `String` is not included because it requires objects, and `SELF_TYPE` is not needed in the absence of objects. To eliminate objects from the language, we need to make two small changes to the Cool typing rules:

1. You can assume that both branches of a conditional expression have the same type (both `Int` or both `Bool`). Therefore, the type of the whole expression will be either `Int` or `Bool`, and it will be the same as the type of the branches. In MP3, you will have to handle the case of *different* types in the branches, that are merged into the "join" of the two types (see Section 7.5 of the Cool manual).
2. A loop expression has type `Int` and evaluates to the value 0. In MP3, you must implement the rule that a loop expression evaluates to a `void` value of type `Object` (Cool manual, Section 7.6). However, for MP2, `Object` is not supported.

The Cool runtime library is not used for MP 2. Also, you cannot use class IO to perform input-output. Instead, you can return a result from the function **main**.

The code generator makes use of the AST constructed in MP1 and static analysis performed by the program *semant*. Your code generator should produce LLVM 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.

This assignment gives you some flexibility in how exactly you generate LLVM code for individual Cool constructs. We will specify certain design choices in order to simplify the project (e.g., how to organize a virtual function table; how to interface to the the external system for printing and I/O). You are responsible for other key design choices, including how to organize the objects of each class, how to perform dynamic dispatch, and how to implement the basic built-in classes (`Int, Bool, String, IO`). Nevertheless, note that there are many key design goals to meet, and there are standard design approaches compilers use to meet these goals. We will discuss these approaches in class or in this handout.

These two assignments will be considerably more difficult and take more time than MP1, so we suggest you get started right away. We have given you a simple exercise to help you learn the LLVM code representation and tools, before writing the code generator. The LLVM Programmer's Manual will also be useful for this purpose.

2 MP2 Distribution

The file *mp2.tar.gz* is available on the course web site, under the Project page. The *mp2* directory contains a README file with documentation on the layout of the skeleton code and the overall structure of the code generator. Read this before you begin writing code.

The subdirectory *mp2/src* contains the skeleton files for the code generation phase, which you will need to modify. You should not need to change any files in any other directories. The mp2/src directory contains:

- *cgen.cc*:
  This file will contain your code generator. We have provided an implementation of some aspects of

code generation; studying this code will help you write the rest of the code generator. It includes a call to code that will build an inheritance graph from the provided AST.

- *cgen.h*:
  This file is the header for the code generator. You may add anything you like to this file. It also provides classes for implementing the inheritance graph.

- *cool-tree.handcode.h*:
  Contains the declarations of classes for AST nodes. You can add field declarations to the classes in *cool-tree.h* by editing *cool-tree.handcode.h*. The macros defined in there are included into *cool-tree.h* during definition.

  So if you want to add a function *void addMe(int)* as pure virtual into for example the *Feature_class* and its implementation into all subclasses, you have to put a definition *virtual void addMe(int)=0* into *Feature_EXTRAS* and *void addMe(int);* into *Feature_SHARED_EXTRAS*. Take a look at *cool-tree.handcode.h* and *cool-tree.h* to fully understand this.

  The definitions of the methods should be added to *cgen.cc*.

- *Makefile*:
  The Makefile for the src/ directory. It is similar to the Makefile for MP1 but has additional targets. You can modify this Makefile (for example, to change the *debug* flag). This file will not be turned in.

- *value_printer.{cc,h}, operand.{cc,h}*: These files contain a small library for printing out simple LLVM instructions, which you may use for the assignment. The provided *cgen.h* includes *value_printer.h*. You may also print your assembly directly. Using the full LLVM API to assemble and pretty-print IR would require some changes to the makefiles - let us know if you want to try this.

- *coolrt.{c,h}*: These files provide an implementation for the COOL runtime library. You will not be using them for MP2.

To compile your code generator for MP2 type *make cgen-1*. Like the Makefile for MP1, the Makefile will compile files from the *cool-support* directory and link the object files into your program.

Note that *cgen.h* and *cgen.cc* use conditional compilation directives (*ifdef* and *ifndef*) to build two different programs, depending on whether the symbol *MP3* is defined. We have set up the Makefile so that when you build *cgen-1*, *MP3* will not be defined, and these regions will not be compiled, nor will they appear in your binary program *cgen-1*. You should not add any code to these sectons for MP2 (you will need to later, for MP3).

The directory *tools-bin* includes the binary tools (lexer, parser, semantic analyzer).

You should also take a short look at the other files in the *cool-support/src*, and *cool-support/include* directories. Especially *cgen_supp.cc* contains general support code for the code generator. You will find a number of handy functions here.

3 Testing the Code Generator

The directory *mp2/test-1/* provides a place for you to test your code generator for MP2. *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 a subset of our own tests.

The directory contains its own Makefile. Some of the targets it provides are:

- `make file.ll`: compile the Cool program file.cl to LLVM assembly
- `make file.bc`: create an LLVM bytecode file from file.ll
- `make file.exe`: create a linked executable from file.bc (this is a no-op for MP 2)
- `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.exe.bc. This is just so you can see whether your code can be optimized effectively by available techniques in LLVM.

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.

In order to run a COOL program and inspect its result, your compiler should add a main() function to the generated LLVM module. This function should call *@Main_main()* and print the result. It should look like the following or equivalent:

```
define i32 @main() {
entry:
        %tpm.0 = call i32 @Main_main(  )
        %tpm.1 = getelementptr [25 x i8], [25 x i8]* @.str, i32 0, i32 0
        %tpm.2 = call i32(i8*, ... ) @printf( i8* %tpm.1, i32 %tpm.0 )
        ret i32 0
}
```

The `.exe` target will fail until your compiler generates a valid assembly file that defines a `main` function with the right signature, and `.out` targets may fail rather than capture output if your generated assembly has sufficiently serious errors.

You should generate this function explicitly using LLVM IR features. To make this easier for you, we've provided a skeleton routine called *CgenClassTable::code_main()* in *cgen.cc*.

Your code generation phase executable *cgen* 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.

4 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; 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 address 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 calls to the function `i8* @malloc(i32)` to allocate heap objects whenever needed, and never free these objects.

5 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 2, you only need to be concerned with Int and Bool values, and only as variables or temporaries (not heap objects).

Here are the guidelines you should follow:

- Values of primitive types should be represented directly as virtual registers (of types `i32` and `i1`) in your generated code, always for MP2 and in most cases for MP3.
- The only time an Int or Bool must live on the heap in your compiler is if an actual object operation needs to be performed on it. Ordinary arithmetic operations (+, <, etc.) are not object operations. Assignment of a value to an Int or Bool variable is not an object operation. There are no object operations in MP2, and few in MP3. (Which are the object operations applicable to Int and Bool?)
- When an Int or Bool value must live on the heap (for an object operation), you must allocate a heap object for it and store the register value into that heap object before the object operation. This is called "boxing" the value.
- When an Int or Bool object value on the heap then needs to be used as a primitive type again (e.g., in an arithmetic operation), you must load the value out of the object into an LLVM register. This is called "unboxing" the value.
- One consequence of this strategy is that MP2 *never* needs an Int or Bool to be on the heap: it can always live in an LLVM register.

- A corollary is that the return value from `@Main_main` must be an `i32` and not `i32*`.
- Think of *let*-variables as names (pointers) for values (objects): this is the correct interpretation for COOL 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. Even if a *let*-variable is of a primitive type (`i32` or `i1`), we will just allocate it on the stack and let *mem2reg* promote it to an SSA register for us. To enable this promotion, all `alloca` instructions should be placed in the entry of the *let*-block.

6 How to attack this project

Since writing a code generator is a fairly big task, we suggest that you take the following steps in order to build your compiler. Make sure to test each portion of code as you complete it.

1. Start by generating the function @main() as decsribed above, so that you can test your compiler even in the early stages of your work.
2. Start by implementing `Int` and `Bool` constants. At first, you can just generate them as the `i32` and `i1` LLVM primitives. *Test your compiler!*
3. Once you have constants, you can implement arithmetic and comparison operators. You can also implement block expressions at this time (e.g., $\{1 + 2; 2 <= 1\}$). *Test your compiler!*.
4. Now try implementing `let`, following the allocation guidelines in the previous section. Use the environment to keep track of the binding from COOL variable names to memory locations (i.e., to LLVM alloca/global/malloc values). *You know what you need to do now!*
5. Next, implement assignment. Here, you will need to think about how the LHS and RHS are implemented, and what should be copied over.
6. Next, tackle `loop` and `if-then-else`. For these, you will need to learn more about LLVM `BasicBlock`s.

   The result of an `if-then-else` is a merge of the results of the two branches. You can allocate an i32 or i1 in the stack (depending on the type of the then-else branches) and then store a different result in each of the branches.
7. 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.

   For MP 2, the only possible error is divide-by-zero. Your program should call the function *abort()* if this happens. We've given you code to insert a declaration for *abort()* in the module.
8. Now test your compiler more thoroughly. You can use the Cool files we will give you in the *test-1* directory, but you should also make your own tests to stress individual cases.

7 What and how to hand in

You have to hand in all files that you modify in this MP. That will include

- *cgen.cc*
- *cgen.h*
- *cool-tree-handcode.h*

You may also wish to modify

- *operand.cc*
- *operand.h*
- *value_printer.cc*
- *value_printer.h*

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.

To hand in the MP, run the hand-in script, *mp2Handin*, located in the directory */class/cs426/cs426/bins* on *EWS* (*linux.ews.illinois.edu*). The script accepts one parameter - your local src folder.

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

When you hand in the MP, your implementation will be tested against the reference implementation for a few tests. This is only meant to ensure that this is a working implementation. It is by no means a complete testing, and is not meant to substitute your own testing.