Lab 1

ECE 479K — Compiler
Spring Semester 2024

Handed out: Jan. 24, 2024 due: Feb. 13, 2024, 11:59 pm

In this assignment and the next (Lab1 and Lab2), you will implement the intermediate code generation phase of your compiler. In Lab1, you'll implement only the language features of Cool shown in the figure below.

1 Cool Language Subset for Lab1

$$
\begin{aligned}
\mathit{program} &::= \mathit{class};\\
\mathit{class} &::= \textbf{class Main } \{\mathit{feature};\}\\
\mathit{feature} &::= \textbf{main}() : \textbf{Int } \{\mathit{expr}\}\\
\mathit{expr} &::= \text{ID} \leftarrow \mathit{expr}\\
&\mid\ \textbf{if } \mathit{expr} \textbf{ then } \mathit{expr} \textbf{ else } \mathit{expr} \textbf{ fi}\\
&\mid\ \textbf{while } \mathit{expr} \textbf{ loop } \mathit{expr} \textbf{ pool}\\
&\mid\ \{\ \mathit{expr};^{+}\ \}\\
&\mid\ \textbf{let } [\![\text{ID} : \text{TYPE } [\leftarrow \mathit{expr}]]\!],^{+}\ s \textbf{ in } \mathit{expr}\\
&\mid\ \mathit{expr} + \mathit{expr}\\
&\mid\ \mathit{expr} - \mathit{expr}\\
&\mid\ \mathit{expr} * \mathit{expr}\\
&\mid\ \mathit{expr}\ /\ \mathit{expr}\\
&\mid\ {\sim}\mathit{expr}\\
&\mid\ \mathit{expr} < \mathit{expr}\\
&\mid\ \mathit{expr} <= \mathit{expr}\\
&\mid\ \mathit{expr} = \mathit{expr}\\
&\mid\ \textbf{not } \mathit{expr}\\
&\mid\ (\mathit{expr})\\
&\mid\ \text{ID}\\
&\mid\ \text{integer}\\
&\mid\ \textbf{true}\\
&\mid\ \textbf{false}
\end{aligned}
$$

Figure 1: Syntax for the subset of Cool to be implemented in Lab1.

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 Lab2, 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 Lab2, you must implement the rule that a loop expression evaluates to a `void` value of type `Object` (Cool manual, Section 7.6). However, for Lab1, `Object` is not supported.

The Cool runtime library is not used for Lab1. 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 Lab0 (the binaries of lexer and parser are provided to you) and static analysis performed by the program `semant`. Your code generator should produce LLVM assembly code that faithfully implements *any* correct Cool program matching the language subset described above. In particular, there is no static error checking in code generation – all erroneous Cool programs have been detected by the front-end phases of the compiler (except for errors that must be detected at run-time, such as divide-by-zero in Lab1).

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 classes in Lab2, and how to return values to the environment in Lab1). You are responsible for other key design choices, including how to implement the basic built-in classes (`Int and Bool` in Lab1). 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.

2 Lab1 Distribution

Lab1 handout 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 directory `lab1_handout/src` contains the skeleton files for code generation that you will need to modify (you should not need to change any files in any other directories), including:

- `cgen.cc`:

  This file will contain your code generator. We have provided some starting code in this file; studying it 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`, `stringtab.handcode.h` :

  The former modifies the declarations of classes for the AST nodes. The latter modifies the declarations of classes `StrTable` and `IntTable`. You can add field declarations to these classes by editing these two files. The macros defined in these files are included into `cool-support/include/{cool_tree.h,stringtab.h}` respectively when building your compiler.

  To know when and how to modify `cool_tree.handcode.h`, you *must* fully understand the inheritance relationship between the various AST node classes. See the documentation at the beginning of `cool_tree.h`.

  So for example, if you want to add a function `Type *get_expr_ty()` to all expressions, you'll need to
  1. declare `virtual op_type get_expr_ty() = 0;` in the macro `Expression_EXTRAS` which in turn gets added `Expression_class`;
  2. declare `op_type get_expr_ty() override;` in `Expression_SHARED_EXTRAS`, which is added to every class that inherits from `Expression_class`, and
  3. go to `cgen.cc` and implement `op_type T::get_expr_ty()` for every `T: Expression_class`.

  Take a look at `cool_tree.handcode.h` and `cool_tree.h` to fully understand this. The definitions (implementations) of the methods should always be added to `cgen.cc`.

- `Makefile`:

  The Makefile for your code generator. `make cgen-1` produces the binary `cgen-1` (codegen-stage-1) for Lab1; `make cgen-2` produces `cgen-2` for Lab2, which you'll not use for now. You can also override the value of `debug` to make debug (default) / release build: `make debug=false cgen-1`. You may modify this Makefile, but it will not be turned in and your compiler must work with the handout version.

- `value_printer.{cc,h}`, `operand.{cc,h}`:

  These files contain a small library for printing out LLVM IR, which you may use for the assignment. The provided `cgen.h` includes `value_printer.h`. You may also print your assembly directly to `std::cout` if you prefer, but you'll have to make sure to get the syntax right.

  Using LLVM's API to construct LLVM IR as in-memory data structures would require substantially more work (esp. for Lab2), so using the provided library is simpler and recommended.

- `coolrt.{c,h}`:

  These files provide a partial implementation of the Cool runtime library for you to complete in Lab2. You will not need them for Lab1.

To compile your code generator for Lab1, type `make cgen-1`. Your program is compiled with `c++17`, so feel free to use the newer features of C++ in your code.

Note that `cgen.h` and `cgen.cc` use conditional compilation directives (`#ifdef LAB2`) to build two different programs depending on if `LAB2` is defined. `make cgen-2` passes `-DLAB2` to `gcc` and `make cgen-1` does not; this is the only difference. For you, this means that you do not need to implement any of the sections with `#ifdef LAB2` in Lab1.

The directory `reference-binaries` includes our binary tools for the previous phases of the Cool compiler (lexer, parser, semantic analyzer). It also contains our code generator for Lab1 called `cgen-1`, which you can use as a reference; it shows you what code to generate for test cases.

You should also take a look at the support code in in the `cool-support/src` and `cool-support/include` directories. You may need to spend most time on `include/cool_tree.h` (esp. the relations between AST node types), then `symtab.h` and `stringtab.h`. You will find a number of handy functions here so that you don't have to reinvent some of the wheels that you need.

3 Testing the Code Generator

The directory `lab1_handout/test` provides a place for you to test your code generator for Lab1. *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 couple of days before the due date, we'll make the some of the test cases available so that you can fix any remaining problems. However, *that will be too late for you to test most of the compiler*, so it is important that you write your own tests for all the features of Cool covered in this lab.

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

- `make <file>.ll`: run your code generator `cgen-1` to compile the Cool program `<file>.cl` to LLVM IR (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, for Lab2);
- `make <file>.out`: execute `<file>.bin` and put the output in `<file>.out`;
- `make clean`: delete all generated files.

This makefile also takes `debug={true|false}` option. When `debug=true`, `cgen-1` will run with `-d`, which sets `cgen_debug` to true in the program. The handout code skeleton adds a few stderr prints when `cgen_debug == true`, and you can add more code to aid your own debugging.

Also note that we optimize your generated program at the highest level LLVM offers (`-O3`). Some incorrect programs may happen to fly under no optimizations, but tend to break down at higher optimization levels. To be sure that you are generating correct LLVM code, you should verify every test case of yours with `make <file>.verify`.

4 Getting Started with LLVM

This Lab and the following Labs make heavy use of LLVM, a compiler development toolchain. Even though in Lab1 and Lab2 you will not use LLVM's API directly, you will output LLVM IR code that conforms to LLVM's language specification and be compiled by `clang` into binary executable. In later Labs, you'll actually use LLVM's API to perform optimizations on LLVM IR code. We strongly recommend that you start to get familiar with LLVM early-on.

- A prebuilt LLVM 15 has been given to you. See the lab environment setup page for details on how to use it. For now you'll at most need to use LLVM tool binaries (`clang`, `opt`, and `lli`) from the command line.

LLVM is an enormous toolchain currently standing at 14 million lines of code, and capable of a wide range of things compiler-related. There is not a single, definitive place to get started with LLVM. For the purposes of our Labs, we can outline some key information you should know and where you can find them:

1. The *intermediate representation* (IR) of LLVM (sometimes also called LLVM bitcode) is key to this lab and the next. Understand what it is, why it is central to LLVM, and how you can produce and consume bitcode with LLVM tools – for example, compiling C++ program to bitcode with `clang`, optimize the bitcode with `opt`, and produce object code from bitcode with `clang` again. This will give you a picture of LLVM's general workflow.
2. Get familiar with the look of LLVM IR: take a piece of LLVM bitcode (for example, use `clang` to compile a toy C program) and read it. What does a function / formal argument / instruction / constant / type look like in LLVM IR?

When in doubt, the LLVM Language Reference is the definitive source of truth for LLVM IR. LLVM's official guide on how to implement your own language may prove more approachable, where you can find digestible pieces of examples in LLVM IR. Introductory LLVM Exercise from previous years may also be helpful, with a simple exercise to get familiar with the IR.

5 Designing the Code Generator

There are many possible ways to write the code generator. One reasonable strategy is to perform code generation in 2 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. Using this information, the second pass recursively walks each feature and generate the LLVM code, respectively, 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. Optimization is mostly the job of the following LLVM passes, and you only need to generate reasonably efficient local code for each tree node.

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:

```
@.str = internal constant [25 x i8] c"Main.main() returned %d\n"

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

```
  %tmp.2 = call i32(i8*, ... ) @printf(i8* %tmp.1, i32 %tmp.0)
  ret i32 0
}
```

LLVM will refuse to compile any invalid bitcode program into binary, so until your compiler generates a valid bitcode program with a `main` function with the right signature, the `<file>.bin` target of the makefile will fail. Any runtime failure of the binary will manifest as `<file>.out` fail.

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`.

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 Lab1, 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 Lab1 and in most cases for Lab2.
- 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 Lab1 (in Lab2, you will use boxing and unboxing for those operations on Int and Bool).
- A corollary is that the return value from `@Main_main` must be an `i32` and not `i32*`.
- Think of `let`-variables as pointers to values (objects): this is the correct interpretation for Cool because the same variable can be assigned different values 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 has 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.

  All `alloca` instructions should be placed in the *entry block* of the function. This placement ensures that your allocation is guaranteed to happen exactly once in each function invocation (even if the `let` is within a loop), improving the ability of optimizations to reason about it.

7 How to attack this project

To simplify your project, we strongly recommend you tackle it incrementally, taking 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, as `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., $\{1 + 2; 2 \leq 1\}$). *Test your compiler!.*

4. Now try implementing `let`, following the allocation guidelines in the previous section. Use `CgenEnvironment` 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 Lab1, 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.

8 What and how to hand in

You will hand in all files under `src/` that you are allowed to modify in this Lab. That is

- `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)

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.

Hand in your files to gradescope. Detailed hand-in instructions will be available when lab DDL appraoches. You can hand in the Lab multiple times. The one we will grade is your last handin.

Do not modify any part of the support code! Your modifications on these files will be ignored. 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 Lab1:

- 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 (≥ 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.

9 Extra Credit Task

20% extra credit: implement Lab1 codegen with LLVM API. The `cgen-1` for extra credit should have the same interface and behavior as base Lab1 described above. You will no longer have access to `value_printer.{cc|h}` and `operands.{cc|h}`, while the other files stay almost the same.

We have provided a different set of code skeleton for Lab1 extra credit including Makefile which automatically includes and links LLVM for you – you can change this Makefile locally but won't submit it. You can find it at `mp1_handout/src_llvm` in the starter code. Some of the routines we predefine for you in Lab1 is written in LLVM there, so these could also be a good place to get started with LLVM API.

A separate submission page will be made availble on gradescope. You only need to submit to one of `Lab1` or `lab1_extra`. (subject to change) We will test this `cgen-1` on the exact same test cases as used for base Lab1.

Extra advice on getting started with LLVM API:

1. Be familiar with the `Value` hierarchy in LLVM: the `Value` class in LLVM API is inherited, directly or indirectly, by almost everything in the IR. `Instruction` inherits from `Value` (why?), and so does `Constant` which in turn `Function` inherits from (again, why?). When you are wondering what class represents a certain piece of code in the IR, you may find it by look at the right place of the `Value` hierarchy.
2. Use the `IRBuilder`, and know how to create and insert different kinds of instructions.
3. LLVM's Code generation to LLVM IR tutorial shows how to translate *Kaleidoscope* into LLVM IR using LLVM API, and it's very relevant for this task.
4. LLVM documentation (auto-generated, such as this one for `llvm::Value`). These are decent at explaining what an LLVM function does, but not helpful the other way around (what API should I use to achieve …).
5. LLVM programmer's manual helps the other way around and is essentially a how-to guide for some aspects LLVM API. This manual only scratches the surface of a few important topics, but still it will take forever to read the whole manual. Instead, *read the table of content* to see what tools are in store in LLVM, and read the specific content on demand.