15-411 Compiler Design, Lab LLVM (Spring 2026)

Seth and co. 🪶

Compilers Due: 11:59 pm, Saturday, March 14, 2026

1 Introduction

The goal of this lab is to compile the L3 language to the LLVM IR. You won't need to add additional language features. Instead, you'll just be adding a new pass to your pre-existing compiler. With this new pass, you'll be able to use the LLVM tools to help debug your compiler.

> **Note:** This lab is meant to take less than a week! We strongly recommend using any extra time to ensure that your compiler is in good shape for Labs 4 and 5. In particular, make sure your register allocator fully works **before** starting Lab 4. It will be much harder to add a register allocator after you've extended your compiler to support memory.

Although we won't require you to do so, we recommend maintaining the LLVM translation pass during Lab 4 and Lab 5. As you're implementing this lab, think about how you can make it extensible to Lab 4 (e.g. handling pointer types). During Lab 5, you can verify your optimizations by outputting LLVM code after each optimization pass.

2 LLVM IR

In this section, we will introduce some key language concepts and useful instructions (though you are free to use any LLVM IR instructions you want).

Function Declarations and Definitions

The LLVM IR program you generate will have both function definitions and function declarations. A function can be defined as follows:

```
define <function type> @<function name>(<arg1 type> %<arg1>, ...) {
    ...
}
```

Each function is defined as a sequence of basic blocks. Each basic block must start with a label and end with a control flow instruction. There must be no non-`phi` instructions between the start of a basic block and the `phi` instructions (i.e. all the `phi` instructions must go first).

A function can be declared as follows:

```
declare <function type> @<function name>(<arg1 type> %<arg1>, ...)
```

Like in Lab 3, non-external functions with name `name` must be called `_c0_name`. To call external functions (such as `abort`), you must explicitly declare them:

```
declare void @abort()
```

Syntax

Here is a (potentially non-exhaustive) list of useful instructions:

Types	
`i1`	type for 1-bit wide integer (used for booleans)
`i32`	type for 32-bit wide integer
Binary operations	
`add` | `sub` | `mul` | `sdiv` | `srem`	arithmetic operations
`and` | `xor` | `or` | `shl` | `ashr`	bitwise operations
Comparison operations	
`slt` | `sgt` | `sle` | `sge` | `eq` | `ne`	$<$ | $>$ | $\leq$ | $\geq$ | $=$ | $\neq$
Expressions	
`<binop> <type> <val 1>, <val 2>`	binary operation
`icmp <op> <type> <val 1>, <val 2>`	comparison operation (returns an `i1` value)
`phi <type> [<val 1>, %<label 1>], [<val 2>, %<label 2>]`	phi move (values must be annotated with the labels for the blocks they originate from)
`call <type> @<function name>(<args>)`	call a non-void function
Statements	
`%<variable name> = <expression>`	assign value to variable
`call void @<function name>(<args>)`	call a void function
Control flow	
`ret <return type> <return value>`	return from a non-void function
`ret void`	return from a void function
`br label %<label>`	jump to a label
`br i1 <val>, label %<label1>, label %<label2>`	jump to label1 if val is true, otherwise jump to label2

Some key things to note:

- Variable names are prefixed with a `%` symbol and function names are prefixed with an `@` symbol. Constants do not need to be prefixed with anything.
- When passing in parameters to a function call, the types of the arguments must be provided.
- There is no move instruction in the LLVM IR (think about why a program in SSA wouldn't need the move instruction). You can replicate it by bitcasting a variable to its own type:

```
%chonk = bitcast i32 %honk to i32
```

- `phi` instructions are parameterized with labels. Each label must correspond to a direct predecessor (i.e. a parent) of the current block.

Example Program

Here is a sample program (featuring our beloved Opal) showcasing some key instructions:

```
declare void @abort()

define i32 @secret_surprise(i32 %best_mascot) {
    start:
        ret i32 %best_mascot
}

define i32 @otters(i32 %opal_father, i32 %opal_mother) {
    start:
        %very_useful_variable = bitcast i1 1 to i1
        br i1 %very_useful_variable, label %label1, label %label2
    label1:
        br label %label3
    label2:
        call void @abort()
        br label %label3
    label3:
        %opal = add i32 %opal_father, %opal_mother
        %surprise = call i32 @secret_surprise(i32 %opal)
        %result = icmp eq i32 %surprise, 411
        br i1 %result, label %label4, label %label5
    label4:
        %a1 = bitcast i32 15411 to i32
        br label %label6
    label5:
        %a2 = bitcast i32 15611 to i32
        br label %label6
```

```
    label6:
        %a3 = phi i32 [%a1, %label4], [%a2, %label5]
        ret i32 %a3
}
```

Additional Resources

To learn more about the LLVM IR, check out the official documentation and this very well-written blog post.

You can see example LLVM IR programs by using Clang to compile a C file to the LLVM IR:

```
clang -emit-llvm -S example.c -o example.ll
```

3 Runtime Environment

Programs compiled with a header file are allowed to assume the declared symbols will be present in the runtime, and that the types declared in the header are accurate to the runtime implementation. The runtime environment will define a function `main()` which calls `_c0_main()`.

We are using LLVM version 15. The `llc` compiler will be used to compile your LLVM IR to assembly. Afterwards, the GNU compiler and linker will be used to link the assembly to the implementations of the external functions. The following commands will be used:

```
llc -march=x86-64 -O0 -regalloc=greedy -optimize-regalloc=true foo.l3.ll
gcc -m64 foo.l3.s ../runtime/run411.c
```

We are running `llc` at `-O0` with register allocation enabled. Note that `llc` will throw an error if your program is not in SSA.

Since division-by-zero is undefined behavior for the `llc` compiler, we will **not** be testing your compiler on tests that should raise `SIGFPE`. However, programs that should raise `SIGABRT` will still be tested. Your test cases will be run with the following flags:

```
../gradecompiler --safe-only -m lab3 --emit=llvm ../tests/foo
```

Your compiler will be tested in an environment that matches the Docker images provided by the course staff.

4 Command-Line Interface

Your compiler is expected to recognize a flag `--emit=llvm` which, when present on the command line, causes your compiler to target the LLVM IR. The file extension should be `.ll`. It is fine if assembly files are also produced when this flag is passed in–those files simply won't be autograded.

Your compiler is still expected to recognize a flag `-t` which stops the compiler immediately after typechecking and before the rest of the compiler runs. The exit code of your compiler should indicate success (0) if the code is well-formed, and failure (1) otherwise. If your compiler indicates success when run with `-t`, then it should be able to compile the file without further errors.

Your compiler should accept an optional command line argument `-l` which must be given the name of a header file as an argument. For instance, we could be calling your compiler using the following command: `bin/c0c --emit=llvm -l ../runtime/15411-l3.h0 $test.l3`. Here, `15411-l3.h0` is a header file.

If your compiler detects any compile-time errors in the source program, it should exit with a non-zero return code. If compilation succeeds and target code is generated, the compiler should then exit with a return code of 0.

5 Testing

Test programs have extension `.l3` and start with one of the following lines:

`//test return` i	program must execute correctly and return i
`//test abort`	program must compile and run but raise `SIGABRT`(6)
`//test error`	program must fail to compile due to an L3 source error
`//test typecheck`	program must typecheck correctly
`//test compile`	program must typecheck, compile, and link

If the test program `$test.l3` is accompanied by a file `$test.h0` (same base name, but `h0` extension), then we compile the test treating `$test.h0` as the header file. Otherwise, we treat `../runtime/15411-l3.h0` as the header file for all `l3` tests, and we will pass that header file to your compiler with the `-l` argument. The `15411-l3.h0` header file describes a library for floating point arithmetic and printing operations. The implementation of this library can be found in `lab3/runtime/run411.c`. Our testing framework will ignore any output performed from the printing operations.

Tests that use a `.h0` header file might typecheck but fail to link because they refer to functions that aren't provided by the system. The header file can even describe a library that can't possibly be implemented (see `busy.l3` and `busy.h0` for an example).

Tests which use a `.h0` header file might also take advantage of functions provided in `libc` or `libgcc` (see `rand.l3` and `rand.h0` for an example). But many of these functions can cause the test cases to behave badly. Therefore, tests that have a custom header file *must* start with the line `//test error` or `//test typecheck`, and will not be executed by the autograding harness.

6 Submission

For this project, you are expected to hand in a complete working compiler for L3 that produces correct target programs written in the LLVM IR. You will submit:

Before Saturday, March 14th, 11:59 pm The complete compiler. You will submit to the **Lab LLVM** assessment on Gradescope. The directory `compiler/lab3` should contain only the

sources for your compiler. **Unlike in previous labs, you do *not* need to make a new folder for the LLVM Lab**. The autograder will build your compiler, run it on the test suite, compile the resulting LLVM IR files with `llc`, link them against our runtime system (if compilation is successful), execute the binaries, and compare the actual and expected results.

A late day is computed based on the active submission on Gradescope. If your active submission is past **11:59 pm** on the due date, it will use a late day.

The files comprising the compiler itself should be collected in a directory `compiler/` which should contain a `Makefile`.

Issuing the shell command

```
% make lab3
```

from within the `compiler` directory should generate the appropriate files so that

```
% bin/c0c --emit=llvm <args>
```

will run your L3 compiler and produce programs written in the LLVM IR. The command

```
% make clean
```

should remove all binaries, heaps, and other generated files.

Autograded Scoring

You may turn in code and have it autograded as many times as you like, without penalty. In fact, we encourage you to hand in to verify that the autograder agrees with the driver results that you use for development, and also as insurance against a last-minute rush. The submission with the highest grade will count.

Your total score for the lab is computed as follows:

$$\text{subtotal} = 25 * \frac{\text{passed basic tests}}{\text{total basic tests}} + 65 * \frac{\text{passed large \& only tests}}{\text{total large \& only tests}}$$

$$\text{total} = \text{subtotal} - (1 \text{ point per compiler failure}) - (0.1 \text{ points per executable timeout})$$