15-411: Compiler Design — Spring 2026

Recitation 6: LLVM & Dynamic Semantics — 27 February

About Code Review Week

As you finish up Lab 3, you should spend some time on improving your codebase and making up for technical debt. If you used any terrible hacks to get register allocation or calling conventions to work, fix them. If you think any part of your code is a jumbled mess, refactor it. You'll want to have a solid base upon which to build Lab 4.

If you haven't already, you should sign up for a code review timeslot using the link on Piazza. Additionally, please also fill in this form with the commit hash or branch that you would like us to review. We will read your code beforehand and ask you questions about it during your team's code review meeting. While we will give you pointers on your style and structure, what we really want to check up on how well you **understand** the code you and your partner have written. We will be using your git commit log to guide our understanding of who implemented what.

If there's any significant section of your compiler that your partner implemented and you did not read, you should read it. If you don't understand how part of your compiler works, you should ask your partner to explain it to you. That said, we don't expect you to remember every detail of your implementation—we just want to make sure that both team members are participating roughly equally and have a thorough understanding of the compiler's structure.

LLVM Basics

Later today, a mini-lab will be released (Lab 3.5?) where you will implement a pass to compile l3 to the LLVM IR, a well-defined intermediate representation supporting an ecosystem of optimization/analysis passes, as well as backends for different target architectures.

A central requirement for the LLVM is static single assignment. Additionally, LLVM is strongly and explicitly typed; each SSA value and function argument is defined with a type, such as `i32` for a 32-bit integer. Finally, SSA values can either have an explicit name or be numbered; both `%opal` and `%6` are both valid names.

Although LLVM is a complex ecosystem, you will only need to use a small subset of its features. A function is defined as follows:

```
define <ret_type> @name(<param_list>) {
  <basic_blocks>
}
```

A key part of the IR is the requirements for basic blocks, which enforce a strict structure for control flow. Firstly, each basic block must begin with a label. Any phi instructions must immediately follow the label, and each phi instruction must have an argument corresponding to each predecessor of the basic block. Additionally, each block must end with exactly one terminal instruction, like a branch or a return.

For example, the following is a valid basic block:

```
BB4:
  %4 = phi i32 [%2, %BB2], [%3, %BB3]
  %5 = add i32 %4, 2
  %6 = mul i32 %5, 7
  ret i32 %6
```

Binary instructions are tagged with the type of the operands, as shown in the example above, and

comparisons are treated this way as well. The statement `%cond = icmp sle i32 1, 2` implies `%cond` is of type `i1`. Finally, branching can be done by conditioning on a variable of type `i1`:

`br i1 %test, label %ltrue, label %lfalse.`

`br label %block` represents an unconditional jump.

Checkpoint 0

The following program contains numerous syntatic issues. Identify them.

```
define i32 @foo(i32 %a, i32 %b) {
entry:
  %c = add %a, %b
  %d = icmp eq i32 %c, 2
  br i1 %d, label %BB1, label %BB2
BB1:
  br label %BB2
BB2:
  %x = phi i32 [%a, %entry]
  ret i32 %d
}
```

Recap: Dynamic Semantics for L2

A configuration of an L2 program could be modeled as one of the two forms

- $\eta \vdash s \blacktriangleright K$ for *executing* statement s
- $\eta \vdash e \triangleright K$ for *evaluating* expression e

where η represents a map from variables to values and K represents the continuation (what to do next with the result of evaluating the current expression, or the next statement).

We're interested in the judgment $c \to c'$, indicating that a configuration c of the form above steps to a configuration c'. Here are a few rules for L2:

$$\begin{aligned} \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K &\longrightarrow \eta \vdash e \triangleright (\mathsf{assign}(x, \_), K) \\ \eta \vdash v \triangleright (\mathsf{assign}(x, \_), K) &\longrightarrow \eta[x \mapsto v] \vdash \mathsf{nop} \blacktriangleright K \\ \eta \vdash \mathsf{nop} \blacktriangleright (s, K) &\longrightarrow \eta \vdash s \blacktriangleright K \end{aligned}$$

We omit many rules—for a more complete set, refer to Lecture 13.

Let c_1 be the initial configuration, and suppose $c_i \to c_{i+1}$. If c_n is a final configuration of the form $\eta \vdash v \triangleright (\mathsf{return}(\_), K)$, then we say that $c_1, c_2, \ldots c_n$ is the *execution trace* of c_1.

Checkpoint 1

Draw the execution trace of configurations starting from:

$$\cdot \vdash \mathsf{seq}(\mathsf{assign}(x, 3), \mathsf{return}(x + 1)) \blacktriangleright \cdot$$

Dynamic Semantics for L3

L3's dynamic semantics is slightly more interesting in that returning from a function call should restore state and control to the configuration prior to the call. We amend our configuration to hold a fourth

element, the call stack S, which consists of tuples of the form $\langle \eta, K \rangle$. We reproduce the rules for single-argument functions below:

$$
\begin{array}{rcl}
S; \eta \vdash f(e) \triangleright K & \longrightarrow & S; \eta \vdash e \triangleright (f(\_), K) \\
S; \eta \vdash v \triangleright (f(\_), K) & \longrightarrow & (S, \langle \eta, K \rangle); [x \mapsto v] \vdash s_f \blacktriangleright \cdot \\
 & & \textit{supposing that } f \textit{ is defined as } f(x)\{s_f;\} \\
(S, \langle \eta, K \rangle); \eta' \vdash v \triangleright (\mathsf{return}(\_), K') & \longrightarrow & S; \eta \vdash v \triangleright K
\end{array}
$$

Checkpoint 2

Draw the execution trace of the following program, starting execution at the beginning of `main`:

```
int f(int x) { return x; }
void g() { 4; }
int main() { int y = f(3); g(); return y; }
```