15-411: Compiler Design — Spring 2026

Recitation 0: Course Introduction — 16 Jan

Welcome to 15-411 Recitation!

In recitation, we will both review what was discussed in lecture and also give you practical tips, tricks, and advice that will help you when implementing your compilers. Please participate and ask questions! We're here to help you succeed.

Course Infrastructure

We'll be giving each of you access to two GitHub repositories:

- `dist`: Starter code, test cases, and tools (read-only)
- <Your team name>: Where you implement your compiler

When you have a ready-to-submit version of your compiler in your team repository, you will submit the corresponding GitHub branch to Gradescope. The autograder will run the corresponding test suites and give you a score. We'll also be using Docker to allow you to run the autograder yourself in an environment identical to that on Gradescope. More detailed instructions will be on the Lab 1 writeup.

Choosing a Language

- OCaml: If you don't have any particular desire to use another language, you should use OCaml. The majority of students choose this language each semester. It's a functional language that is very similar to SML but has many more standard libraries that make compiler implementation easier.
- Rust: It has native support for functional features while being a systems language that you can really optimize. However, Rust may not be as easy to pick up as OCaml, so we recommend using it only if you have some prior experience.
- Other: If you choose another language, make sure that you are very familiar with it and that it has the libraries and features you need to properly implement your compiler.

Collaboration

Here are a few tips for getting started collaborating on Lab 1 with your partner:

- You absolutely need to set up at least two meetings per week with your partner. Expect each of the meetings to last at least two hours. Use this time to discuss the overall architecture of your implementation and divide up the work.
- In your first meeting, you should spend a lot of time talking about your collaboration styles and making plans of attack.
- Use GitHub pull requests to read and review your partner's code. One partner makes a pull request, and the other reads it, makes comments, and eventually merges it into master.
- It might be tempting to divide the work between frontend and backend, but this is usually a bad idea because (1) the backend will require more work than the frontend and (2) it will be harder for both partners to gain knowledge of the entire system.
- Since Lab 1 is significantly easier than the later labs, you are **required** to implement an efficient register allocator. You'll save time later on and be able to focus on other parts of the compiler. Lab 1 checkpoint is intended to help you with this.

x86 Assembly - Resources

The backend of your compiler will demand a lot of work with x86 assembly. There are a lot of resources out there to help you with assembly, so we encourage you to search around! We also recommend these sites:

- https://www.felixcloutier.com/x86/: A very good instruction reference.
- https://godbolt.org/: gcc compiler explorer that lets you input C code, and shows you the assembly produced by gcc. Very useful for debugging the assembly produced by your own compiler!

Liveness Analysis

The first step in assigning registers is to determine which temps interfere with each other. If a temp is defined on line ℓ, it interferes with all variables in liveout(ℓ). We use the following rules to construct live-in and live-out sets:

$$\text{LiveIn}(\ell) = \text{Uses}(\ell) \cup (\text{LiveOut}(\ell) - \text{Defs}(\ell))$$

$$\text{LiveOut}(\ell) = \bigcup_{s \in \text{succ}(\ell)} \text{LiveIn}(s)$$

The following is an program in 3-address abstract assembly.

```
1  main:
2    x <- 42
3    t1 <- 2
4    t2 <- x % t1
5    if (t2 == 0) then goto L1 else goto L2
6  L1:
7    x <- x + 1
8    z <- 1
9    goto L3
10 L2:
11   t3 <- 1
12   z <- t3 * -1
13   goto L3
14 L3:
15   %eax <- z * x
16   return
```

Checkpoint 0

Analyze the above program to determine the live-out and live-in sets at each of the lines. Then draw the interference graph.

Spilling

Recall that we assign registers by coloring the interference graph (we'll see an algorithm for this in class on Tuesday). However, oftentimes the number of colors required to color the graph is larger than the maximum number of registers we have available. When this occurs, we have to choose some temps to store on the stack. This is called **spilling**.

A naive strategy is to color the graph with as many colors as required, and then arbitrarily choose colors to be stored on the stack until the rest can be stored in registers. However, this can be optimized.

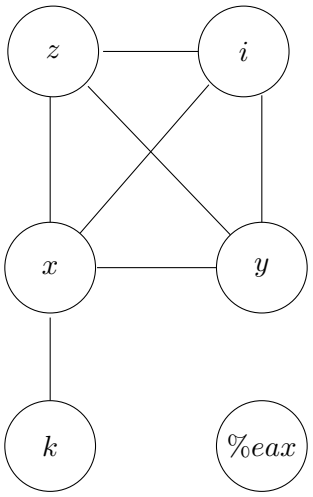
Checkpoint 1

Brainstorm some relevant features to consider when deciding which temps to spill.

Checkpoint 2

The following is a sample program in 3-address abstract assembly, along with its interference graph.

```
1   main:
2     i <- 1
3     z <- 2
4     x <- z + 1
5     y <- z + 2
6     if (i < 5) then goto L1 else goto L2
7   L1:
8     i <- i + 1
9     x <- y * x
10    if (i < 5) then goto L1 else goto L2
11  L2:
12    x <- x + z
13    k <- y * 2
14    x <- x + k
15    %eax <- x
16    return
```



If we had 3 registers, which variables could we spill in order to make the interference graph 3-colorable? Which of these is probably the best choice?

Lab 1 Tip: Spilling Temps

We can't fit all of our data in registers, so we spill into memory. But we need at least one operand in a register for most arithmetic operations. This is getting into the software engineering part of the course, but we will outline one strategy that you can use.

You will need to reserve a register, typically `%r11d`. Perform register allocation, then scan through your instructions looking for memory-memory operations. You then insert a `mov` from the destination to `%r11d`, perform the operation, then move `%r11d` back to memory.

In a functional language, you can implement this in a pass similar to code generation, where you case on instruction type and produce either a list with the input instruction, or a list with the moves into and out of `%r11d`.

(Bonus) Inductive Definitions and Inference Rules

As you'll soon see, it's convenient to define the rules and structures of a programming language inductively. For example, you could say that

> "If a is an expression and b is an expression, then $\texttt{plus}(a, b)$ is also an expression."
> "Integer contants are expressions."

and use these rules (and others) to build up an entire mathematical expression.

To make it easier to express more complex inductive systems, such as an entire programming language, we express inference rules using the following form:

$$\frac{J_1 \;\; J_2 \;\; \ldots \;\; J_n}{J}$$

We call $J_1 \ldots J_n$ and J *judgments*. $J_1 \ldots J_n$ are the *premises* of the rule and J is the *conclusion*. (Sometimes you may see more than one conclusion; this is shorthand for two rules with the same premises). In this form, we could express our example rules from above as

$$\frac{a \text{ exp} \quad b \text{ exp}}{\texttt{plus}(a, b) \text{ exp}} \; A_1 \qquad \frac{n \in \mathbb{Z}}{n \text{ exp}} \; A_2$$

Checkpoint 3

Given `zero` to denote the number 0 and $\texttt{succ}(n)$ to denote the successor of n, write two rules that inductively define the judgment "n nat" to describe the natural numbers.

Now, write two more rules that inductively define the judgment "$\text{sum}(a, b) = c$" to mean that the sum of the natural numbers a and b is equal to c.