15-411: Compiler Design — Fall 2017

Recitation 0: Course Introduction — 01 September

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.

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 0

Given $\texttt{zero}$ 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.

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

We'll set up Git hooks on your team repository so that whenever someone pushes to a submission branch (lab1, test1, lab2, test2, etc.) the autograder will run the corresponding test suite and give you a score. More details will be on the Lab 1 writeup (coming Tuesday).

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.
- Haskell: If you're more interested in theory than systems, want to learn more about monads and typeclasses, and are lazy, you should use Haskell. Like Ocaml, it is a functional language with a rich standard library, but you may have more difficulty if you've never used Haskell before.
- SML: If you don't want to learn the minor differences between SML and Ocaml, and you don't mind the lack of standard libraries, you should use SML.
- Swift: If you're not particularly comfortable with functional programming (the idea of writing thousands of lines of code without a single `for` loop scares you) or you're excited to try something that's new this semester, you should use Swift.

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, it might be a good idea to implement an initial version of a register allocator. You'll save time later on and be able to focus on other parts of the compiler.