Lecture 7 — Functions taking/returning functions
Today

- Course motivation/overview
- Begin first-class functions
Why these 3?

<table>
<thead>
<tr>
<th></th>
<th>dynamically typed</th>
<th>statically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional</td>
<td>Scheme</td>
<td>SML</td>
</tr>
<tr>
<td>object-oriented</td>
<td>Ruby</td>
<td>Java</td>
</tr>
</tbody>
</table>

- ML: polymorphic types complementary to OO-style subtyping, rich module system for abstract types, and rich pattern-matching.

- Scheme: dynamic typing, “good” macros, fascinating control operators (may skip), and a minimalist design.

- Ruby: classes but not types, a more complete commitment to OO, mixins.

Runners-up: Haskell (laziness & purity), Prolog (unification & backtracking), Smalltalk (even more OO than Ruby), ...
Are these useful?

The way we use ML/Scheme/Ruby in 341 can make them seem almost “silly” precisely because we focus on *interesting language concepts*

“Real” programming needs file I/O, string operations, floating-point, graphics libraries, project managers, unit testers, threads, foreign-function interfaces, ...

- These languages have all that and more!
- If we used Java the same way in 341, Java would seem “silly” too
First-Class Functions

• Functions are values.
  (Variables in the environment are bound to them.)

• We can pass functions to other functions.
  – *Factor* common parts and *abstract* different parts.

• Most polymorphic functions take functions as arguments.
  – Non-example: `fun f x = (x,2,x)`

• Some functions taking functions are not polymorphic.
Type Inference and Polymorphism

ML can infer function types based on function bodies. Possibilities:

- The argument/result must be one specific type.
- The argument/result can be any type, but may have to be the same type as other parts of argument/result.
- “equality types” (see last week’s section)

We will study this parametric polymorphism more later.

Without it, ML would be a pain (e.g., a different list library for every list-element type).

Fascinating: If \( f: \text{int} \rightarrow \text{int} \), there are lots of values \( f \) could return. If \( f: \text{’a} \rightarrow \text{’a} \), whenever \( f \) returns, it returns its argument!
Anonymous Functions

As usual, we can write functions anywhere we write expressions.

- We already could:
  
  \[
  \text{(let fun f x = e in f end)}
  \]

- Here is a more concise way (better style when possible):
  
  \[
  \text{(fn x => e)}
  \]

- Cannot do this for recursive functions (why?)
Returning Functions

Syntax note: \textasciitilde \textasciitilde \textasciitilde “associates to the right”

- \texttt{t1\textasciitilde t2\textasciitilde t3} means \texttt{t1\textasciitilde (t2\textasciitilde t3)}

Again, there is nothing new here.

The key question: What about \textit{free variables} in a function value? What \textit{environment} do we use to \textit{evaluate} them?

Are such free variables useful?

You must understand the answers to move beyond being a novice programmer.