CSE 341:
Programming Languages

Dan Grossman

Winter 2008

Lecture 7— Functions taking/returning functions

Today

- A little more on course motivation/overview
- Begin first-class functions

Why these 3?

- 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.

Runners-up: Haskell (laziness & purity), Prolog (unification & backtracking), Smalltalk (OO like Ruby), ...

	dynamically typed	statically typed
functional	Scheme	SML
object-oriented	Ruby	Java

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 Java were in 341, it 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 = 42`
- 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.
- Some hand-waving about "equality types"

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:int->int`, there are lots of values `f` could return. If `f:'a->'a`, whenever `f` returns, it returns its argument!

Anonymous Functions

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

- We already could:

```
(let fun f x = e in f end)
```

- Here is a more concise way (better style when possible):

```
(fn x => e)
```

- Cannot do this for recursive functions (why?)

Returning Functions

Syntax note: `->` "associates to the right"

- `t1->t2->t3` means `t1->(t2->t3)`

Again, there is nothing new here.

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

Are such free variables useful?

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