Higher-order programming

- Higher-order programming, e.g., with `map` and `filter`, is great
- Language support for closures makes it very pleasant
- Without closures, we can still do it more manually / clumsily
  - In OOP (e.g., Java) with one-method interfaces
  - In procedural (e.g., C) with explicit environment arguments
- Working through this:
  - Shows connections between languages and features
  - Can help you understand closures and objects

Example in ML

datatype 'a mylist = Cons of 'a * ('a mylist) | Empty
(* ('a -> 'b) -> 'a mylist -> 'b mylist *)
fun map f xs = case xs of ...
(* ('a -> bool) -> 'a mylist -> 'a mylist *)
fun filter f xs = case xs of ...
(* 'a mylist -> int *)
fun length xs = case xs of ...

val doubleAll = map (fn x => x*2)
val countNs xs = length (filter (fn x => x=n) xs)

Java

- Java 8 likely to have closures (like C#, Scala, Ruby, ...)
  - Write like `lst.map((x) => x.age)`
    `.filter((x) => x > 21)`
    `.length()`
  - Make parallelism and collections much easier
  - Encourage less mutation
  - Hard parts for language designers:
    - Implementation with other features and VM
    - Evolving current standard library (else not worth it?)
- But how could we program in an ML style in Java today...
  - Won’t look like pseudocode above
  - Was even more painful before Java had generics

One-method interfaces

interface Func<B,A> { B m(A x); }
interface Pred<A> { boolean m(A x); }
interface Foo { String m(int x, int y); }

- An interface is a named type [constructor]
- An object with one method can serve as a closure
  - Different instances can have different fields [possibly different types] like different closures can have different environments [possibly different types]
- So an interface with one method can serve as a function type

List types

Creating a generic list class works fine
- Assuming `null` for empty list here, a choice we may regret
- `null` makes every type an option type with implicit `valOf`
Higher-order functions

- Let's use static methods for `map`, `filter`, `length`
- Use our earlier generic interfaces for “function arguments”
- These methods are recursive
  - Less efficient in Java 😊
  - Much simpler than common previous-pointer acrobatics

```java
static <A,B> List<B> map(Func<B,A> f, List<A> xs){
    if(xs==null) return null;
    return new List<B>(f.m(xs.head), map(f,xs.tail));
}
static <A> List<A> filter(Pred<A> f, List<A> xs){
    if(xs==null) return null;
    if(f.m(xs.head))
        return new List<A>(xs.head), filter(f,xs.tail);
    return filter(f,xs.tail);
}
static <A> length(List<A> xs){ ... }
```

Why not instance methods?

A more OO approach would be instance methods:

```java
class List<T> {
    <B> List<B> map(Func<B,T> f){...}
    List<T> filter(Pred<T> f){...}
    int length(){...}
}
```

Can work, but interacts poorly with null for empty list
- Cannot call a method on null
- So leads to extra cases in all clients of these methods if a list might be empty

An even more OO alternative uses a subclass of List for empty-lists rather than null
- Then instance methods work fine!

Clients

- To use `map` method to make a `List<Bar>` from a `List<Foo>`:
  - Define a class `C` that implements `Func<Bar,Foo>`
  - Use fields to hold any “private data”
  - Make an object of class `C`, passing private data to constructor
  - Pass the object to `map`
- As a convenience, can combine all 3 steps with anonymous inner classes
  - Mostly just syntactic sugar
  - But can directly access enclosing fields and final variables
  - Added to language to better support callbacks
- Syntax an acquired taste? See lec11.java

Now C [for C experts]

- In Java, objects, like closures, can have “parts” that do not show up in their types (interfaces)
- In C, a function pointer is just a code pointer, period
  - So without extra thought, functions taking function pointer arguments won’t be as useful as functions taking closures
- A common technique:
  - Always define function pointers and higher-order functions to take an extra, explicit environment argument
  - But without generics, no good choice for type of list elements or the environment
  - Use `void*` and various type casts...

The C trick

[ignore if not (yet) a C wizard; full implementation in lec11.c]

Don’t do this:

```c
list_t* map(void* (*f)(void*), list_t xs){
    ... f(xs->head) ...
}
```

Do this to support clients that need private data:

```c
list_t* map(void* (*f)(void*,void*), void* env, list_t xs) {
    ... f(env, xs->head) ...
}
```

List libraries like this aren’t common in C, but callbacks are!
- Lack of generics means lots of type casts in clients 😊