University of Washington CSE 341: Programming Languages
Unit 3 Reading Notes

Standard Description: This summary covers **some of** *the same material as lecture and section. It can help to read about the material in a narrative style and to have the material for an entire unit of the course in a single document, especially when reviewing the material later. Please report errors in these notes, even typos. This summary is not a sufficient substitute for attending class, reading the associated code, etc.*

Last updated: January 31, 2023

Contents

Introduction and Some Terms

This unit focuses on *first-class functions* and *function closures*. By "first-class" we mean that functions can be computed, passed, stored, etc. *wherever* other values can be computed, passed, stored, etc. As examples, we can pass them to functions, return them from functions, put them in pairs, have them be part of the data a variant carries, etc. "Function closures" refers to functions that use variables defined outside of them, which makes first-class functions much more powerful, as we will see after starting with simpler first-class functions that do not use this ability. The term *higher-order function* just refers to a function that takes or returns other functions.

Terms like first-class functions, function closures, and higher-order functions are often confused with each other or considered synonyms. Because so much of the world is not careful with these terms, we will not be too worried about them either. But the idea of first-class functions and the idea of function closures really are distinct *concepts* that we often use together to write elegant, reusable code. For that reason, we will delay the idea of closures, so we can introduce it as a separate concept.

There is an even more general term, *functional programming*. This term also is often used imprecisely to refer to several distinct concepts. The two most important and most common are:

- Not using mutable data in most or all cases: We have avoided mutation throughout the course so far and will mostly continue to do so.
- Using functions as values, which is what this unit is all about

There are other things that are also considered related to functional programming:

- A programming style that encourages recursion and recursive data structures
- Programming with a syntax or style that is closer to traditional mathematical definitions of functions
- Anything that is not object-oriented programming (this one is really incorrect)
- Using certain programming idioms related to *laziness*, a technical term for a certain kind of programming construct/idiom we will study, briefly, later in the course

An obvious related question is "what makes a programming language a functional language?" Your instructor has come to the conclusion this is not a question for which there is a precise answer and barely makes sense as a question. But one could say that a functional language is one where writing in a functional style (as described above) is more convenient, more natural, and more common than programming in other styles. At a minimum, you need good support for immutable data, first-class functions, and function closures. More and more we are seeing new languages that provide such support but also provide good support for other styles, like object-oriented programming.

Taking Functions as Arguments

The most common use of first-class functions is passing them as arguments to other functions, so we motivate this use first.

Here is a first example of a function that takes another function:

```
let rec n_times (f,n,x) =
  if n = 0
  then x
  else f (n_times (f,n-1,x))
```

We can tell the argument `f` is a function because the last line calls `f` with an argument. What `n_times` does is compute `f(f(...(f(x))))` where the number of calls to `f` is `n`. That is a genuinely useful helper function to have around. For example, here are 3 different uses of it:

```
let double x = x+x
let x1 = n_times (double,4,7) (* answer: 112 *)

let increment x = x+1
let x2 = n_times (increment,4,7) (* answer: 11 *)

let x3 = n_times (List.tl,2,[4,8,12,16]) (* answer: [12,16] *)
```

Like any helper function, `n_times` lets us *abstract* the common parts of multiple computations so we can *reuse* some code in different ways by passing in different arguments. The main novelty is making one of those arguments a function, which is a powerful and flexible programming idiom. It also makes perfect sense — we are not introducing any new language constructs here, just using ones we already know in ways you may not have thought of.

Once we define such abstractions, we can find additional uses for them. For example, even if our program today does not need to triple any values n times, maybe tomorrow it will, in which case we can just define the function `triple_n_times` using `n_times`:

```
let triple x = 3*x

let triple_n_times (n,x) = n_times (triple,n,x)
```

We can also put functions in data structures. For example, we could make a list of functions. Since OCaml lists require all elements to have the same type, that means the functions in any particular list will need to all have the same type. For example here is a list of type `(int -> int) list` with two elements:

```
let double x = x * 2
let incr   x = x + 1
let funcs = [double; incr]
```

What could we do with such a list? One idea is to take some value and apply all the functions in a list of any length to it in succession:

```
let rec apply_funcs (fs,x) =
  match fs with
  | [] -> x
  | f :: fs' -> apply_funcs (fs', f x)
```

For example:

- `apply_funcs (funcs, 100)` evaluates to 201
- `apply_funcs (List.rev funcs, 100)` evaluates to 202
- `apply_funcs (funcs @ funcs, 100)` evaluates to 403.

Polymorphic Types and Functions as Arguments

Let us now consider the type of the `n_times` example above, which is `('a -> 'a) * int * 'a -> 'a`. It might be simpler at first to consider the type `(int -> int) * int * int -> int`, which is how `n_times` is used for `x1` and `x2` above: It takes 3 arguments, the first of which is itself a function that takes and returns an `int`. Similarly, for `x3` we use `n_times` as though it has type `(int list -> int list) * int * int list -> int list`. But choosing either one of these types for `n_times` would make it less useful because only some of our example uses would type-check. The type `('a -> 'a) * int * 'a -> 'a` says the third argument and result can be any type, but they have to be the *same* type, as does the argument and return type for the first argument. When types can be any type and do not have to be the same as other types, we use different letters (`'b`, `'c`, etc.)

This is called *parametric polymorphism*, or perhaps more commonly *generic types*. It lets functions take arguments of any type. It is a separate issue from first-class functions:

- There are functions that take functions and do not have polymorphic types
- There are functions with polymorphic types that do not take functions.

However, many of our examples with first-class functions will have polymorphic types. That is a good thing because it makes our code more reusable.

Without parametric polymorphism, we would have to redefine lists for every type of element that a list might have. Instead, we can have functions that work for any kind of list, like `length`, which has type `'a list -> int` even though it does not use any function arguments. Conversely, here is a higher-order function that is not polymorphic: it has type `(int->int) * int -> int`:[1]

```
let rec times_until_zero (f,x) =
  if x = 0 then
    0
  else
    1 + times_until_zero(f, f x)
```

Anonymous functions

There is no reason that a function like `triple` that is passed to another function like `n_times` needs to be defined at top-level. As usual, it is better style to define such functions locally if they are needed only locally. So we could write:

```
let triple_n_times (n,x) =
  let triple x = 3*x in
  n_times (triple,n,x)
```

In fact, we could give the `triple` function an even smaller scope: we need it only as the first argument to `n_times`, so we could have a let-expression there that evaluates to the triple function:

```
let triple_n_times (n,x) =
  n_times((let triple y = 3*y in triple), n, x)
```

Notice that in this example, which is actually poor style, we need to have the let-expression "return" `triple` since, as always, a let-expression produces the result of the expression after the `in`. In this case, we simply look up `triple` in the environment, and the resulting function is the value that we then pass as the first argument to `n_times`.

[1]It would be better to make this function tail-recursive using an accumulator.

OCaml has a much more concise way to define functions right where you use them, as in this final, best version:

```
let triple_n_times (n,x) = n_times ((fun y -> 3*y), n, x)
```

This code defines an *anonymous function* `fun y -> 3*y`. It is a function that takes an argument `y` and has body `3*y`. The `fun` is a keyword and `->` (not `=`) is also part of the syntax. We never gave the function a name (it is *anonymous*, see?), which is convenient because we did not need one. We just wanted to pass a function to `n_times`, and in the body of `n_times`, this function is bound to `f`.

It is common to use anonymous functions as arguments to other functions. Moreover, you can put an anonymous function anywhere you can put an expression — it simply is a value, the function itself. The only thing you cannot do with an anonymous function is recursion, exactly because you have no name to use for the recursive call. In such cases, you need to use `let` to define the function as before.

You could use anonymous functions with variable bindings. For example, these two bindings are exactly the same thing:

```
let increment x = x + 1
let increment = fun x -> x+1
```

They both bind `increment` to a value that is a function that returns its argument plus 1. So function-bindings like the first line are syntactic sugar. As usual, the version with the sugar is better style but it helps to understand that they can be defined in terms of more basic things, in this case a variable binding to an anonymous function.

Unnecessary Function Wrapping

While anonymous functions are incredibly convenient, there is one poor idiom where they get used for no good reason. Consider:

```
let nth_tail_poor (n,x) = n_times ((fun ys -> List.tl ys), n, x)
```

What is `fun ys -> List.tl ys`? It is a function that returns the list-tail of its argument. But there is already a variable bound to a function that does the exact same thing: `List.tl`! In general, there is no reason to write `fun x -> f x` when we can just use `f`. This is analogous to the beginner's habit of writing `if x then true else false` instead of `x`. Just do this:

```
let nth_tail (n,x) = n_times (List.tl, n, x)
```

Maps and filters

We now consider a very useful higher-order function over lists:

```
let rec map (f,xs) =
  match xs with
  | [] -> []
  | x::xs' -> (f x) :: (map (f,xs'))
```

The `map` function takes a list and a function `f` and produces a new list by applying `f` to each element of the list. Here are two example uses:

```
let x1 = map (increment, [4,8,12,16]) (* answer: [5,9,13,17] *)
let x2 = map (hd, [[1,2],[3,4],[5,6,7]]) (* answer: [1,3,5] *)
```

The type of `map` is illuminating: `('a -> 'b) * 'a list -> 'b list`. You can pass `map` any kind of list you want, but the argument type of `f` must be the element type of the list (they are both `'a`). But the return type of `f` can be a different type `'b`. The resulting list is a `'b list`. For `x1`, both `'a` and `'b` are *instantiated* with `int`. For `x2`, `'a` is `int list` and `'b` is `int`.

The OCaml standard library provides a very similar function `List.map`, but it is defined in a curried form, a topic we will discuss later in this unit.

The definition and use of `map` is an incredibly important idiom even though our particular example is simple. We could have easily written a recursive function over lists of integers that incremented all the elements, but instead we divided the work into two parts: The map *implementer* knew how to traverse a recursive data structure, in this case a list. The map *client* knew what to do with the data there, in this case increment each number. You could imagine either of these tasks — traversing a complicated piece of data or doing some calculation for each of the pieces — being vastly more complicated and best done by different developers without making assumptions about the other task. That is exactly what writing `map` as a helper function that takes a function lets us do.

Here is a second very useful higher-order function for lists. It takes a function of type `'a -> bool` and an `'a list` and returns the `'a list` containing only the elements of the input list for which the function returns true:

```
let rec filter (f,xs) =
  match xs with
  | [] -> []
  | x::xs' -> if f x then
                x :: (filter (f,xs'))
              else
                filter (f,xs')
```

Here is an example use that assumes the list elements are pairs with second component of type `int`; it returns the list elements where the second component is even:

```
let get_all_even_snd xs = filter((fun (_,v) -> v mod 2 = 0), xs)
```

(Notice how we are using a pattern for the argument to our anonymous function.)

Returning functions

Functions can also return functions. Here is an example:

```
let double_or_triple f =
  if f 7 then
    fun x -> 2*x
  else
    fun x -> 3*x
```

The type of `double_or_triple` is `(int -> bool) -> (int -> int)`: The if-test makes the type of `f` clear and as usual the two branches of the if must have the same type, in this case `int->int`. However, OCaml will print the type as `(int -> bool) -> int -> int`, which is the same thing. The parentheses are unnecessary because the `->` "associates to the right", i.e., `t1 -> t2 -> t3 -> t4` is `t1 -> (t2 -> (t3 -> t4))`.

Not just for numbers and lists

Because ML programs tend to use lists a lot, you might forget that higher-order functions are useful for more than lists. Some of our first examples just used integers. But higher-order functions also are great for our own data structures. Here we use an `is_even` function to see if all the constants in an arithmetic expression are even. We could easily reuse `true_of_all_constants` for any other property we wanted to check.

```
type exp = | Constant of int | Negate of exp | Add of exp * exp | Multiply of exp * exp

let is_even v =
  (v mod 2 = 0)

let rec true_of_all_constants (f,e) =
  match e with
  | Constant i -> f i
  | Negate e1 -> true_of_all_constants (f,e1)
  | Add(e1,e2) -> true_of_all_constants (f,e1) && true_of_all_constants (f,e2)
  | Multiply(e1,e2) -> true_of_all_constants (f,e1) && true_of_all_constants (f,e2)

let all_even e = true_of_all_constants (is_even,e)
```

Lexical Scope

So far, the functions we have passed to or returned from other functions have been *closed*: the function bodies used only the function's argument(s) and any locally defined variables. But we know that functions can do more than that: they can use any bindings that are in scope. Doing so in combination with higher-order functions is very powerful, so it is crucial to learn effective idioms using this technique. But first it is even more crucial to get the semantics right. This is probably the most subtle and important concept in the entire course, so go slowly and read carefully.

*The body of a function is evaluated in the environment where the function is **defined**, not the environment where the function is **called**.* Here is a very simple example to demonstrate the difference:

```
let x = 1
let f y = x + y
let x = 2
let y = 3
let z = f (x+y)
```

In this example, `f` is bound to a function that takes an argument `y`. Its body also looks up `x` in the environment where `f` was defined. Hence this function *always* increments its argument since the environment at the definition maps `x` to 1. Later we have a different environment where `f` maps to this function, `x` maps to 2, `y` maps to 3, and we make the call `f (x+y)`. Here is how evaluation proceeds:

- Look up `f` to get the previously described function.
- Evaluate the argument `x+y` in the *current* environment by looking up `x` and `y`, producing 5.
- Call the function with the argument 5, which means evaluating the body `x+y` in the *"old"* environment where `x` maps to 1 extended with `y` mapping to 5. So the result is 6.

Notice the argument was evaluated in the current environment (producing 5), but the function body was evaluated in the "old" environment. We discuss below why this semantics is desirable, but first we define this semantics more precisely and understand the semantics with additional silly examples that use higher-order functions.

This semantics is called *lexical scope*. The alternate, inferior semantics where you use the current environment (which would produce 7 in the above example) is called *dynamic scope*.

Environments and Closures

We have said that functions are values, but we have not been precise about what that value exactly is. We now explain that a function value has *two parts*, the *code* for the function (we clearly need the code) and the *environment that was current when we created the function*. These two parts really do form a "pair" but we put "pair" in quotation marks because it is not an OCaml pair, just something with two parts. You *cannot* access the parts of the "pair" separately; all you can do is call the function. This call uses both parts because it evaluates the code part using the environment part.

This "pair" is called a *function closure* or just *closure*. The reason is that while the code itself can have *free variables* (variables that are not *bound* inside the code so they need to be bound by some outer environment), the closure carries with it an environment that provides all these bindings. So the closure overall is "closed" — it has everything it needs to produce a function result given a function argument.

In the example above, the binding `let f y = x + y` bound `f` to a closure. The code part is the function `fun y -> x + y` and the environment part maps `x` to 1. Therefore, any call to this closure will return `y+1`.

(Silly) Examples Including Higher-Order Functions

Lexical scope and closures get more interesting when we have higher-order functions, but the semantics already described will lead us to the right answers.

Example 1:

```
let x = 1
let f y =
  let x = y+1 in
  fun q -> x + y + q
let x = 3
let g = f 4
let y = 5
let z = g 6
```

Here, `f` is bound to a closure where the environment part maps `x` to 1. So when we later evaluate `f 4`, we evaluate `let x = y+1 in fun q -> x + y + q` in an environment where `x` maps to 1 extended to map `y` to 4. But then due to the let-expression in the function body we shadow `x` so we evaluate `fun q -> x + y + q` in an environment where `x` maps to `5` and `y` maps to 4. How do we evaluate a function like `fun q -> x + y + q`? We create a closure with the current environment. So `f 4` returns a closure that, when called, will always add 9 to its argument, no matter what the environment is at any call-site. Hence, in the last line of the example, `z` will be bound to 15.

Example 2:

```
let f g =
  let x = 3 in
  g 2
let x = 4
let h y = x + y
let z = f h
```

In this example, `f` is bound to a closure that takes another function `g` as an argument and returns the result of `g 2`. The closure bound to `h` *always* adds 4 to its argument because the argument is `y`, the body is `x+y`, and the function is defined in an environment where `x` maps to 4. So in the last line, `z` will be bound to 6. The binding `let x = 3` is totally irrelevant: the call `g 2` is evaluated by looking up `g` to get the closure that was passed in and then using that closure with *its environment* (in which `x` maps to 4) with 2 for an argument.

Why Lexical Scope

While lexical scope and higher-order functions take some getting used to, decades of experience make clear that this semantics is what we want. Much of the rest of this course unit will describe various widespread idioms that are powerful and that rely on lexical scope.

But first we can also motivate lexical scope by showing how dynamic scope (where you just have one current environment and use it to evaluate function bodies) leads to some fundamental problems.

First, suppose in Example 1 above the body of `f` was changed to `let w = y+1 in fun q -> w + y + q`. Under lexical scope this is fine: we can always change the name of a local variable and its uses without it affecting anything. Under dynamic scope, now the call to `g 6` will make no sense: we will try to look up `w`, but there is no `w` in the environment at the call-site.

Second, consider again the original version of Example 1 but now change the line `let x = 3` to `let x = "hi"`. Under lexical scope, this is again fine: that binding is never actually used. Under dynamic scope, the call to `g 6` will look-up `x`, get a string, and try to add it, which should not happen in an OCaml program that type-checks.

Similar issues arise with Example 2: The body of `f` in this example is awful: we have a local binding we never use. Under lexical scope we can remove it, changing the body to `g 2` and know that this has no effect on the rest of the program. Under dynamic scope it would have an effect. Also, under lexical scope we *know* that any use of the closure bound to `h` will add 4 to its argument regardless of how other functions like `g` are implemented and what variable names they use. This is a key separation-of-concerns that only lexical scope provides.

For “regular” variables in programs, lexical scope is the way to go. There are some compelling uses for dynamic scope for certain idioms, but few languages have special support for these (Racket does) and very few if any modern languages have dynamic scope as the default. But you have seen one feature that is more like dynamic scope than lexical scope: exception handling. When an exception is raised, evaluation has to "look up" which handle expression should be evaluated. This "look up" is done using the dynamic call stack, with no regard for the lexical structure of the program.

Passing Closures to Iterators Like Filter

The examples above are silly, so we need to show useful programs that rely on lexical scope. The first idiom we will show is passing functions to iterators like *map* and *filter*. The functions we previously passed did not use their environment (only their arguments and maybe local variables), but being able to pass in closures makes the higher-order functions much more widely useful. Consider:

```
let rec filter (f,xs) =
  match xs with
  | [] -> []
  | x::xs' -> if f x then x::(filter (f,xs')) else filter (f,xs')

let allGreaterThanSeven xs = filter (fun x -> x > 7, xs)

let allGreaterThan (xs,n) = filter (fun x -> x > n, xs)
```

Here, `allGreaterThanSeven` is "old news" — we pass in a function that removes from the result any numbers 7 or less in a list. But it is much more likely that you want a function like `allGreaterThan` that takes the "limit" as a parameter `n` and uses the function `fun x -> x > n`. Notice this requires a closure and lexical scope! When the implementation of `filter` calls this function, we need to look up `n` in the environment where `fun x -> x > n` was defined.

Here are two additional examples:

```
let allShorterThan1 (xs,s) = filter (fun x -> String.length x < String.length s, xs)

let allShorterThan2 (xs,s) =
  let i = String.length s in
  filter (fun x -> String.length x < i, xs)
```

Both these functions take a list of strings `xs` and a string `s` and return a list containing only the strings in `xs` that are shorter than `s`. And they both use closures, to look up `s` or `i` when the anonymous functions get called. The second one is more complicated but a bit more efficient: The first one recomputes `String.size s` once per element in `xs` (because `filter` calls its function argument this many times and the body evaluates `String.size s` each time). The second one "precomputes" `String.size s` and binds it to a variable `i` available to the function `fun x -> String.size x < i`.

Fold and More Closure Examples

Beyond map and filter, a third incredibly useful higher-order function is *fold*, which can have several slightly different definitions and is also known by names such as *reduce* and *inject*. Here is one common definition of `fold_left`, which processes its list argument left-to-right (as expected, one could also define a `fold_right`):

```
let rec fold_left (f,acc,xs) =
  match xs with
  | [] -> acc
  | x::xs' -> fold_left (f, f(acc,x), xs')
```

`fold_left` takes an "initial answer" `acc` and uses `f` to "combine" `acc` and the first element of the list, using this as the new "initial answer" for "folding" over the rest of the list. We can use `fold_left` to take care of iterating over a list while we provide some function that expresses how to combine elements. For example, to sum the elements in a list `foo`, we can do:

```
fold_left ((fun (x,y) -> x+y), 0, foo)
```

As with `map` and `filter`, much of `fold_left`'s power comes from clients passing closures that can have "private fields" (in the form of variable bindings) for keeping data they want to consult. Here are two examples. The first counts how many elements are in some integer range. The second checks if all elements are strings shorter than some other string's length.

```
let numberInRange (xs,lo,hi) =
  fold_left ((fun (x,y) -> x + (if y >= lo && y <= hi then 1 else 0)), 0, xs)

let areAllShorter (xs,s) =
  let i = String.size s in
  fold_left ((fun (x,y) -> && String.size y < i), true, xs)
```

This pattern of splitting the recursive traversal (`fold_left` or `map`) from the data-processing done on the elements (the closures passed in) is fundamental. In our examples, both parts are so easy we could just do the whole thing together in a few simple lines. More generally, we may have a very complicated set of data structures to traverse or we may have very involved data processing to do. It is good to *separate these concerns* so that the programming problems can be solved separately.

Introducing More Closure Idioms

We have now covered the *semantics* of function closures, and we have seen how useful this semantics is for the functions passed to iterators like `map`, `filter`, and `fold_left`. There are various other useful iterator-like higher-order functions, and such functions are useful for a variety of data structures. But iterators are far from the only powerful use of function closures.

So most of the rest of this unit is some additional common, important idioms that use closures. These idioms are just some of the more common ones and we give just a few examples. Remember idioms are just some of the more common uses of language features, worth recognizing because they are common. However, the first idiom, currying, is so common in OCaml that it (a) has built-in syntactic sugar and (b) is the default approach in the standard library and the generally preferred style for functions that conceptually take multiple arguments.

As we present these idioms, we will digress for a couple of other OCaml language features we need to discuss because they come up: OCaml's support for mutable data and OCaml's "value restriction" which is an unfortunate but necessary restriction on type-checking that sometimes arises when using higher-order functions.

Closure Idiom: Currying and Partial Application

We have already seen that in OCaml every function takes exactly one argument, so you have to use an idiom to get the effect of multiple arguments. Our previous approach passed a tuple as the one argument, so each part of the tuple is conceptually one of the multiple arguments. Another more clever and often more convenient way is to have a function take the first conceptual argument and return another function that takes the second conceptual argument and so on. Lexical scope is essential to this technique working correctly.

This technique is called *currying* after a logician named Haskell Curry who studied related ideas (so if you do not know that, then the term currying does not make much sense).

Defining and Using a Curried Function

Here is a silly example of a "three argument" function that uses currying:

```
let sorted3 = fun x -> (fun y -> (fn z -> (z >= y && y >= x)))
```

(The parentheses are unnecessary, but included in case they help understand what the nested function bodies are.) If we call `sorted3 4` we will get a closure that has `x` in its environment. If we then call this closure with `5`, we will get a closure that has `x` and `y` in its environment. If we then call this closure with `6`, we will get `true` because 6 is greater than 5 and 5 is greater than 4. That is just how closures work.

So `((sorted3 4) 5) 6` computes exactly what we want and feels pretty close to calling `sorted3` with 3 arguments. Even better, the parentheses in these nested function calls are also optional, so we can write exactly the same thing as `sorted3 4 5 6`, which is actually fewer characters than our old tuple approach where we would have:

```
let sorted3_tupled (x,y,z) = z >= y && y >= x
let someClient = sorted3_tupled (4,5,6)
```

In general, the syntax `e1 e2 e3 e4` is implicitly the nested function calls `(((e1 e2) e3) e4)` and this choice was made because it makes using a curried function so pleasant.

Partial Application

Even though we might expect most clients of our curried `sorted3` to provide all three conceptual arguments, they might provide fewer and use the resulting closure later. This is called "partial application" because we are providing a subset (more precisely, a prefix) of the conceptual arguments. As a silly example, `sorted3 0 0` returns a function that returns `true` if its argument is nonnegative.

Syntactic Sugar for Curried Functions

There is syntactic sugar for defining curried functions; you can just separate the conceptual arguments by spaces rather than using anonymous functions. So the better style for our `sorted3` function would be:

```
let sorted3 x y z = z >= y && y >= x
```

In general each curried argument can be a pattern.

Anonymous functions can also use this syntactic sugar, so for example:

```
fun x y -> x + y
```

is better style than the identical:

```
fun x -> fun y -> x + y
```

Many people learn to program in OCaml just knowing that:

- Multiple arguments can be separated by spaces
- Functions can be partially applied, producing functions that take the remaining arguments

This is true, but on a more funadmental level it is just function closures with some syntactic sugar, which in general is:

- `let [rec] f p1 ... pn = e` means `let [rec] f = fun p1 -> fun p2 -> ... fun pn -> e`
- `fun p1 ... pn -> e` means `fun p1 -> fun p2 -> ... fun pn -> e`

Partial Application and Higher-Order Functions

Currying is particularly convenient for creating similar functions with iterators. For example, here is a curried version of a `fold_left` function for lists (where, incidentally the function passed for `f` also takes its two arguments via the currying idiom rather than taking a pair). We show the definition without and then with the syntactic sugar; the latter is better.

```
let rec fold_left f = fun acc -> fun xs ->
  match xs with
  | [] -> acc
  | x::xs' -> ((fold_left f) ((f acc) x)) xs'

let rec fold_left f acc xs =
  match xs with
  | [] -> acc
  | x::xs' -> fold_left f (f acc x) xs'
```

Now we could use this fold to define a function that sums a list elements like this, where again we first show a version without syntactic sugar:

```
let sum1 xs = ((fold_left (fun x -> fun y -> x+y)) 0) xs

let sum1 xs = fold_left (fun x y -> x+y) 0 xs
```

But even the version with syntactic sugar is unnecessarily complicated compared to just using partial application:

```
let sum2 = fold_left (fun x y -> x+y) 0
```

`sum1` is implemented with unnecessary function wrapping where the function we are wrapping is the result of `fold_left (fun x y -> x+y)`. Because `sum2` does not, it is better style (and more efficient because we evaluate `fold_left (fun x y -> x+y) 0` once when creating the binding for `sum2`).

Because currying is the normal approach to (simulating) multiargument functions in OCaml, we get used to seeing currying in standard-library functions and generally use the same approach for our own functions. While this is particularly useful for higher-order functions, it is common for all functions. For example, here are some types from the standard-library for lists:

```
val List.compare_lengths : 'a list -> 'b list -> int = <fun>
val List.combine : 'a list -> 'b list -> ('a * 'b) list = <fun>
val List.map = ('a -> 'b) -> 'a list -> 'b list = <fun>
val List.filter = ('a -> bool) -> 'a list -> 'a list = <fun>
val List.fold_left = ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Partial application can be useful even when not using higher-order functions. In this example, both `zip` and `range` are defined with currying and `countup` partially applies `range`. The `add_numbers` function turns the list `[v1,v2,...,vn]` into `[(1,v1),(2,v2),...,(n,vn)]`.

```
let rec zip xs ys =
  match (xs,ys) with
  | ([],[]) -> []
  | (x::xs',y::ys') -> (x,y) :: (zip xs' ys')
  | _ -> failwith "empty"

let rec range i j = if i > j then [] else i :: range (i+1) j

let countup  = range 1 (* partial application *)

let add_numbers xs = zip (countup (length xs)) xs
```

Currying vs. Tupling

The convenience of partial application is why OCaml's standard library and the common style in OCaml is to use currying instead of tupling for basically all functions.

We did tupling first just so that we could see both idioms. OCaml programmers generally prefer currying, given the convenient syntax and the opportunity for partial application. It happens to be implemented more efficiently too, but the difference rarely matters and that's an implementation detail that one would have no way of knowing based on the language definition.

There are situations where tupling is more convenient and powerful. In particular, we saw in the previous unit that a function taking a tuple can be passed the tuple result of another function. This technique does not work with curried functions.

Note tupling and currying are different techniques and it will be a type-checking error, sometimes a difficult to diagnose one, if you pass multiple arguments in a tuple to a function expecting curried arguments or vice versa.

Closure Idiom: Combining Functions

Function composition

When we program with lots of functions, it is useful to create new functions that are just combinations of other functions. You have probably done similar things in mathematics, such as when you compose two functions. For example, here is a function that does exactly function composition (where from now on we will generally use currying for multiple arguments):

```
let compose f g x = f (g x)
```

This may be easier to understand with a bit less syntactic sugar as the typical use-case would be to take two functions `f` and `g` and produce a function taking `x`:

```
let compose f g = fun x -> f (g x)
```

In either case, `compose` takes two functions `f` and `g` and returns a function that applies its argument to `g` and makes that the argument to `f`. Crucially, the code `fun x -> f (g x)` uses the `f` and `g` in the environment where it was defined, i.e., this works due to the semantics of lexical scope.

Notice the type of `compose` is inferred to be `('a -> 'b) * ('c -> 'a) -> 'c -> 'b`, which is equivalent to what you might write: `('b -> 'c) * ('a -> 'b) -> ('a -> 'c)` since the two types simply use different type-variable names consistently.

We can use `compose` like this:

```
let double_plus_one = compose (fun x -> x + 1) (fun x -> 2 * x)
let seven = double_plus_one 6
```

As a separate-but-convenient syntactic issue, OCaml lets us define *infix* operators that start with various non-letter symbols. So we could also do this:

```
let (%) = compose
let double_plus_one = (fun x -> x + 1) % (fun x -> 2 * x)
let seven = double_plus_one 6
```

Here is a perhaps more compelling example of defining a function that computes the square-root of the absolute value of an integer by converting it to a floating-point number:

```
let sqrt_of_abs = Float.sqrt % float_of_int % Int.abs
```

We can compare this version to a couple other correct implementations. First there is nothing wrong with this direct implementation:

```
let sqrt_of_abs i = Float.sqrt (float_of_int (Int.abs i))
```

But this version does not communicate directly that `sqrt_of_abs` is simply two function compositions. Second we can consider:

```
let sqrt_of_abs i = (Float.sqrt % float_of_int % Int.abs) i
```

This is another example of unnecessary function wrapping, in this case the function produced by the function compositions.

The Pipeline Operator

Like function composition in mathematics, our `%` operators has the strange-to-many-programmers property that the computation proceeds from right-to-left: "Take the absolute value, convert it to a real, and compute the square root" may be easier to understand than, "Take the square root of the conversion to real of the absolute value."

We can define convenient syntax for left-to-right as well. In fact, this is in the OCaml standard-library, but its instructive to show the incredibly short definition. Let's first define our own infix operator that lets us put the function to the right of the argument we are calling it with:

```
let (|>) x f = f x
```

Now we can write:

```
let sqrt_of_abs i = i |> abs |> Real.fromInt |> Math.sqrt
```

This operator, commonly called the *pipeline operator*, has become popular in several functional programming languages in recent years. There is nothing complicated about the semantics of the pipeline operator — it is a higher-order function that reverses the order you write down a function and its argument.

Here is a final example of combining functions. Here we take f and g where f and g produce options We do not call g if f produces None and we call g with v if f produces—Some v—. As is often the case with higher-order functions, the type of pipeline_option gives a strong hint as to its behavior.

```
let pipeline_option (f, g) =
  fun x ->
    match f x with
    | None -> None
    | Some y -> g y

let sqrt_if_positive =
  pipeline_option ((fun i -> if i > 0 then Some (float_of_int i) else None),
                   (fun x -> Some x) % sqrt)

let _ = sqrt_if_positive 3
let _ = sqrt_if_positive (-3)
```

Combining Functions to Curry and Uncurry Other Functions

Sometimes functions are curried but the arguments are not in the order you want for a partial application. Or sometimes a function is curried when you want it to use tuples or vice-versa. Fortunately, we use the idea of combining functions to take functions using one approach and produce functions using another:

```
let swap_curry1 f = fun x -> fun y -> f y x
let swap_curry2 f x y = f y x
let curry_tupled f x y = f (x,y)
let tuple_curried f (x,y) = f x y
let swap_tupled f (x,y) = f (y,x)
```

Looking at the types of these functions can help you understand what they do. They may "seem backward" but reading them carefully can straighten us out. For example consider tuple_curried. It takes a function f and returns a function that takes a pair (x,y). The body of that pair-taking function calls f with two curried arguments. So overall, the point is to call tuple_curried with one argument, a curried function, and get back a function that takes a pair. That pair-taking function calls the curried f with the pair's components.

As an aside, the types are also fascinating because if you pronounce -> as "implies" and * as "and", the types of all the functions above are logical tautologies.

The Value Restriction

Once you have learned currying and partial application, you might try to use it to create a polymorphic function. Unfortunately, certain uses, such as this one, do not work in OCaml:

```
let remove_empty_lists = List.filter (fun x -> List.length x > 0)
```

Given what we have learned so far, there is no reason why this should not work. The function should take a list of lists (since we apply `List.length` to list elements, the elements must themselves be lists) so the type would be `'a list list -> 'a list list`. But in fact the REPL reports an odd `'weak1 list list -> '_weak1 list list` and matters get weirder. If we then do `remove_empty_lists [["hi"]]`, then this works but `remove_empty_lists` now has type `string list list -> string list list` and so another use like `remove_empty_lists [[3]]` does not type-check.

It would perhaps be less confusing if OCaml simply would not type-check `remove_empty_lists` at all, but they have added some special "look to see if it is only used for one particular type" support for convenience, and the actual type-checking details get complicated and less elegant.

In any case, it should surprise us that we do not get the basic polymorphic type `'a list list -> 'a list list`. The reason is called the *value restriction*, and it is sometimes annoying. It is in the language for good reason: without it, the type-checker might allow some code to break the type system. This can happen only with code that is using mutation and the code above is not using mutation, but the type-checker does not know that. We can explain this in more detail in the next unit after we study type inference.

For now, the simplest approach is to ignore this issue until and unless it arises for you. We are bringing it up just in case you do trip across it. The value restriction is this: For a binding like `remove_empty_lists` to have a polymorphic type (one with `'a` in it), the expression for it must be a value (or variable), but `List.filter (fun x -> List.length x > 0)`, is not a value, it is a function call. Hence the type for `remove_empty_lists` cannot be polymorphic, so the type-checker will first use this `'_weak1` stuff but later based on how it is used, pick the first non-polymorphic type it can.

If you really need a polymorphic type, then the easiest solution is to do what we have otherwise said not to do: wrap the expression in a function like this:

```
let remove_empty_lists xs = List.filter (fun x -> List.length x > 0) XS
```

Because this is syntactic sugar for

```
let remove_empty_lists = fun xs -> List.filter (fun x -> List.length x > 0) xs
```

and because functions *are* values, the value restriction no longer prevents giving `remove_empty_lists` a polymorphic type. Unless necessary for the value restriction, do not do such function wrapping.

Mutation via OCaml References

We now finally introduce OCaml's support for mutation since we will use it for the next callback idiom. Mutation is okay in some settings. A key approach in functional programming is to use it only when "updating the state of something so all users of that state can see a change has occurred" is the natural way to model your computation. Moreover, we want to keep features for mutation separate so that we know when mutation is not being used.

In OCaml, most things really cannot be mutated. Instead you must create a *reference*, which is a container whose *contents* can be changed. You create a new reference with the expression `ref e` (the initial contents are the result of evaluating `e`). You get a reference `r`'s current contents with `!r` (not to be confused with negation in Java or C), and you change `r`'s contents with `r := e`. The type of a reference that contains values of type `t` is written `t ref`. This is a *different* type from `t`.

One good way to think about a reference is as a record with one field where that field can be updated with the := operator. This is, in fact, exactly how it is actually defined in OCaml.

Here is a short example:

```
let x = ref 0
let x2 = x (* x and x2 both refer to the same reference *)
let x3 = ref 0
(* let y = x + 1*) (* wrong: x is not an int, so does not type-check *)
let y = (!x) + 1 (* y is 1 *)
let _ = x := (!x) + 7 (* the contents of the reference x refers to is now 7 *)
let z1 = !x  (* z1 is  7 *)
let z2 = !x2 (* z2 is also 7 -- with mutation, aliasing matters*)
let z3 = !x3 (* z3 is 0 *)
```

Closure Idiom: Callbacks

The next common idiom we consider is implementing a library that detects when "events" occur and informs clients that have previously "registered" their interest in hearing about events. Clients can register their interest by providing a "callback" — a function that gets called when the event occurs. Examples of events for which you might want this sort of library include things like users moving the mouse or pressing a key. Data arriving from a network interface is another example. Computer players in a game where the events are "it is your turn" is yet another.

The purpose of these libraries is to allow multiple clients to register callbacks. The library implementer has no idea what clients need to compute when an event occurs, and the clients may need "extra data" to do the computation. So the library implementor should not restrict what "extra data" each client uses. A closure is ideal for this because a function's type `t1 -> t2` does not specify the types of any other variables a closure uses, so we can put the "extra data" in the closure's environment.

If you have used "event listeners" in Java's Swing library, then you have used this idiom in an object-oriented setting. In Java, you get "extra data" by defining a subclass with additional fields. This can take an awful lot of keystrokes for a simple listener, which is a (the?) main reason the Java language added anonymous inner classes (which you do not need to know about for this course, but we will show an example later), which are closer to the convenience of closures. Similarly, the way JavaScript supports adding code to web pages that respond to changes or user actions is also through using functions or objects as callbacks.

In OCaml, we will use mutation to show the callback idiom. This is reasonable because we really do want registering a callback to "change the state of the world" — when an event occurs, there are now more callbacks to invoke.

Our example uses the idea that callbacks should be called when a key on the keyboard is pressed. We will pass the callbacks an `int` that encodes which key it was. Our interface just needs a way to register callbacks. (In a real library, you might also want a way to unregister them.)

```
val onKeyEvent : (int -> unit) -> unit = <fun>
```

Clients will pass a function of type `int -> unit` that, when called later with an `int`, will do whatever they want. To implement this function, we just use a reference that holds a list of the callbacks. Then when an event actually occurs, we assume the function `onEvent` is called and it calls each callback in the list:

```
let cbs : (int -> unit) list ref = ref []
let onKeyEvent f = cbs := f::(!cbs) (* The only "public" binding *)
let onEvent i =
   let rec loop fs =
     match fs with
     | [] -> ()
     | f::fs' -> (f i; loop fs')
    in loop (!cbs)
```

Most importantly, the type of `onKeyEvent` places no restriction on what extra data a callback can access when it is called. Here are different clients (calls to `onKeyEvent`) that use different bindings of different types in their environment. (The `let _ = e` idiom is common for executing an expression just for its side-effect, in this case registering a callback.)

```
let timesPressed = ref 0
let _ = onKeyEvent (fun _ -> timesPressed := (!timesPressed) + 1)

let print_if_pressed i =
    onKeyEvent (fun j -> if i=j
                         then print_endline ("pressed " ^ string_of_int i)
                         else ())

let _ = print_if_pressed 4
let _ = print_if_pressed 11
let _ = print_if_pressed 23
```

Optional: Closure Idiom: Abstract Data Types

This last closure idiom we will consider is the fanciest and most subtle. It is not the sort of thing programmers typically do — there is usually a simpler way to do it in a modern programming language. It is included as an advanced example to demonstrate that a record of closures that have the same environment is a lot like an object in object-oriented programming: the functions are like methods and the bindings in the environment are like private fields and methods. There are no new language features here, just lexical scope. It suggests (correctly) that functional programming and object-oriented programming are more similar than they might first appear (a topic we will revisit later in the course; there are also important differences).

The key to an abstract data type (ADT) is requiring clients to use it via a collection of functions rather than directly accessing its private implementation. Thanks to this abstraction, we can later change how the data type is implemented without changing how it behaves for clients. In an object-oriented language, you might implement an ADT by defining a class with all private fields (inaccessible to clients) and some public methods (the interface with clients). We can do the same thing in OCaml with a record of closures; the variables that the closures use from the environment correspond to the private fields.

As an example, consider an implementation of a set of integers that supports creating a new bigger set and seeing if an integer is in a set. Our sets are mutation-free in the sense that adding an integer to a set produces a new, different set. (We could just as easily define a mutable version using OCaml's references.) In OCaml, we could define a type that describes our interface:

```
type set = S of {
  insert : int -> set;
  member : int -> bool;
  size : unit -> int
}
```

Roughly speaking, a set is a record with three fields, each of which holds a function. It would be simpler to write:

```
type set = {
  insert : int -> set;
  member : int -> bool;
  size : unit -> int
}
```

but this does not work in OCaml because by default record types cannot be recursive (a compiler flag can affectthis). So we will instead deal with the mild inconvenience of having a constructor `S` around our record of functions defining a set even though sets are each-of types, not one-of types. Notice we are not using any new types or features; we simply have a type describing a record with fields named `insert`, `member`, and `size`, each of which holds a function.

Once we have an empty set, we can use its `insert` field to create a one-element set, and then use that set's `insert` field to create a two-element set, and so on. So the only other thing our interface needs is a binding like this:

```
let empty_set = ... : set
```

Before implementing this interface, let's see how a client might use it (many of the parentheses are optional but may help understand the code):

```
let use_sets () =
  let S s1 = empty_set in
  let S s2 = s1.insert 34 in
  let S s3 = s2.insert 34 in
  let S s4 = s3.insert 19 in
  if s4.member 42 then
    99
  else if s4.member 19 then
    17 + s3.size ()
  else
    0
```

Again we are using no new features. `s1.insert` is reading a record field, which in this case produces a function that we can then call with 34. The `let` bindings use pattern-matching to "strip off" the `S` constructors on values of type `set`.

There are many ways we could define `empty_set`; they will all use the technique of using a closure to "remember" what elements a set has. Here is one way:

```
let empty_set =
  let rec make_set xs = (* xs is "private field" in result *)
    let contains i = List.mem i xs in (* contains is "private method" *)
    S {
        insert = (fun i -> if contains i then
                             make_set xs
```

```
                            else
                              make_set (i :: xs));
        member = contains;
        size   = (fun () -> List.length xs)
      }
  in
  make_set []
```

All the fanciness is in `make_set`, and `empty_set` is just the record returned by `make_set []`. What `make_set` returns is a value of type `set`. It is essentially a record with three closures. The closures can use `xs`, the helper function `contains`, and `make_set`. Like all function bodies, they are not executed until they are called.

Optional: Closures in Other Languages

To conclude our study of function closures, we digress from OCaml to show similar programming patterns in Java (using generics and interfaces) and C (using function pointers taking explicit environment arguments). This is optional material you can skip, and you should if you are not familiar with Java or C. However, it may help you understand closures by seeing similar ideas in other settings, and it should help you see how central ideas in one language can influence how you might approach problems in other languages. That is, it could make you a better programmer in Java or C.

For both Java and C, we will "port" this OCaml code, which defines our own polymorphic linked-list type constructor and three polymorphic functions (two higher-order) over that type. We will investigate a couple ways we could write similar code in Java or C, which will can help us better understand similarities between closures and objects (for Java) and how environments can be made explicit (for C). In OCaml, there is no reason to define our own type constructor since `'a list` is already written, but doing so will help us compare to the Java and C versions.

```
type 'a mylist = Cons of 'a * ('a mylist) | Empty

let rec map f xs =
  match xs with
  | Empty -> Empty
  | Cons(x,xs) -> Cons(f x, map f xs)

let rec filter f xs =
  match xs with
  | Empty -> Empty
  | Cons(x,xs) -> if f x then Cons(x,filter f xs) else filter f xs

let rec length xs =
  match xs with
  | Empty -> 0
  | Cons(_,xs) -> 1 + length xs
```

Using this library, here are two client functions. (The latter is not particularly efficient, but shows a simple use of `length` and `filter`.)

```
let doubleAll = map (fun x -> x * 2)
let countNs xs n = length (filter (fun x -> x=n) xs)
```

Optional: Closures in Java using Objects and Interfaces

Note: This section could use updating. In recent CSE 341 offerings, we have sometimes shown some of this at the end of the course and have changed the example slightly and including versions using Java's support for closures.

Since version 8, Java includes support for closures much like most other mainstream object-oriented languages now do (C#, Scala, Ruby, ...), but it is worth considering how we might write similar code in Java without this support, as was necessary for almost two decades. While we do not have first-class functions, currying, or type inference, we do have generics (Java did not used to) and we can define interfaces with one method, which we can use like function types. Without further ado, here is a Java analogue of the code, followed by a brief discussion of features you may not have seen before and other ways we could have written the code:

```
interface Func<B,A> {
    B m(A x);
}
interface Pred<A> {
    boolean m(A x);
}
class List<T> {
    T       head;
    List<T> tail;
    List(T x, List<T> xs) {
        head = x;
        tail = xs;
    }
    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> int length(List<A> xs) {
        int ans = 0;
        while(xs != null) {
            ++ans;
            xs = xs.tail;
        }
        return ans;
    }
}

class ExampleClients {
    static List<Integer> doubleAll(List<Integer> xs) {
        return List.map((new Func<Integer,Integer>() {
                             public Integer m(Integer x) { return x * 2; }
```

```
                         }),
                        xs);
    }
    static int countNs(List<Integer> xs, final int n) {
        return List.length(List.filter((new Pred<Integer>() {
                                             public boolean m(Integer x) { return x==n; }
                                        }),
                                        xs));
    }
}
```

This code uses several interesting techniques and features:

- In place of the (inferred) function types `'a -> 'b` for `map` and `'a -> bool` for `filter`, we have generic interfaces with one method. A class implementing one of these interfaces can have fields of any types it needs, which will serve the role of a closure's environment.
- The generic class `List` serves the role of the variant type binding. The constructor initializes the `head` and `tail` fields as expected, using the standard Java convention of `null` for the empty list.
- Static methods in Java can be generic provided the type variables are explicitly mentioned to the left of the return type. Other than that and syntax, the `map` and `filter` implementations are similar to their ML counterparts, using the one method in the `Func` or `Pred` interface as the function passed as an argument. For `length`, we could use recursion, but choose instead to follow Java's preference for loops.
- If you have never seen anonymous inner classes, then the methods `doubleAll` and `countNs` will look quite odd. Somewhat like anonymous functions, this language feature lets us create an object that implements an interface without giving a name to that object's class. Instead, we use `new` with the interface being implemented (instantiating the type variables appropriately) and then provide definitions for the methods. As an inner class, this definition can use fields of the enclosing object or *final* local variables and parameters of the enclosing method, gaining much of the convenience of a closure's environment with more cumbersome syntax. (Anonymous inner classes were added to Java to support callbacks and similar idioms.)

There are many different ways we could have written the Java code. Of particular interest:

- Tail recursion is not as efficient as loops in implementations of Java, so it is reasonable to prefer loop-based implementations of `map` and `filter`. Doing so without reversing an intermediate list is more intricate than you might think (you need to keep a pointer to the previous element, with special code for the first element). The recursive version is easy to understand, but would be unwise for very long lists.
- A more object-oriented approach would be to make `map`, `filter`, and `length` instance methods instead of static methods. The method signatures would change to:

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

  The disadvantage of this approach is that we have to add special cases in any *use* of these methods if the client may have an empty list. The reason is empty lists are represented as `null` and using `null` as the receiver of a call raises a `NullPointerException`. So methods `doubleAll` and `countNs` would have to check their arguments for `null` to avoid such exceptions.

- Another more object-oriented approach would be to not use `null` for empty lists. Instead we would have an abstract list class with two subclasses, one for empty lists and one for nonempty lists. This approach is a much more faithful object-oriented approach to variant types with multiple constructors, and using it makes the previous suggestion of instance methods work out without special cases. It does seem more complicated and longer to programmers accustomed to using `null`.
- Anonymous inner classes are just a convenience. We could instead define "normal" classes that implement `Func<Integer,Integer>` and `Pred<Integer>` and create instances to pass to `map` and `filter`. For the `countNs` example, our class would have an `int` field for holding n and we would pass the value for this field to the constructor of the class, which would initialize the field.

Optional: Closures in C Using Explicit Environments

C does have functions, but they are not closures. If you pass a pointer to a function, it is only a code pointer. As we have studied, if a function argument can use only its arguments, higher-order functions are much less useful. So what can we do in a language like C? We can change the higher-order functions as follows:

- Take the environment explicitly as another argument.
- Have the function-argument also take an environment.
- When calling the function-argument, pass it the environment.

So instead of a higher-order function looking something like this:

```
int f(int (*g)(int), list_t xs) { ... g(xs->head) ... }
```

we would have it look like this:

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

We use `void*` because we want `f` to work with functions that use environments of different types, so there is no good choice. Clients will have to cast to and from `void*` from other compatible types. We do not discuss those details here.

While the C code has a lot of other details, this use of explicit environments in the definitions and uses of `map` and `filter` is the key difference from the versions in other languages:

```
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

typedef struct List list_t;
struct List {
  void * head;
  list_t * tail;
};
list_t * makelist (void * x, list_t * xs) {
  list_t * ans = (list_t *)malloc(sizeof(list_t));
  ans->head = x;
  ans->tail = xs;
  return ans;
}
list_t * map(void* (*f)(void*,void*), void* env, list_t * xs) {
  if(xs==NULL)
    return NULL;
  return makelist(f(env,xs->head), map(f,env,xs->tail));
}
list_t * filter(bool (*f)(void*,void*), void* env, list_t * xs) {
  if(xs==NULL)
    return NULL;
  if(f(env,xs->head))
    return makelist(xs->head, filter(f,env,xs->tail));
  return filter(f,env,xs->tail);
}
int length(list_t* xs) {
  int ans = 0;
  while(xs != NULL) {
    ++ans;
    xs = xs->tail;
  }
  return ans;
}
void* doubleInt(void* ignore, void* i) { // type casts to match what map expects
  return (void*)(((intptr_t)i)*2);
}
list_t * doubleAll(list_t * xs) { // assumes list holds intptr_t fields
  return map(doubleInt, NULL, xs);
}
bool isN(void* n, void* i) { // type casts to match what filter expects
  return ((intptr_t)n)==((intptr_t)i);
}
int countNs(list_t * xs, intptr_t n) { // assumes list hold intptr_t fields
  return length(filter(isN, (void*)n, xs));
}
```

As in Java, using recursion instead of loops is much simpler but likely less efficient. Another alternative would be to define structs that put the code and environment together in one value, but our approach of using an extra `void*` argument to every higher-order function is more common in C code.

For those interested in C-specification details: Also note the client code above, specifically the code in functions `doubleInt`, `isN`, and `countNs`, is not portable because it is not technically legal to assume that an `intptr_t` can be cast to a `void*` and back unless the value started as a pointer (rather than a number that fits in an `intptr_t`). While the code as written above is a fairly common approach, portable versions would either need to use a pointer to a number or replace the uses of `void*` in the library with `intptr_t`. The latter approach is still a reusable library because any pointer can be converted to `intptr_t` and back.