1 Function composition

**The composition operator.** 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:

```plaintext
fun compose (f,g) = fn x => f (g x)
```

It takes two functions \( f \) and \( g \) and returns a function that applies its argument to \( g \) and makes that the argument to \( f \). Notice the type of \( \text{compose} \) is inferred to be \((\text{a} \to \text{b}) \times (\text{c} \to \text{a}) \to \text{c} \to \text{b}\). You might have expected \((\text{b} \to \text{c}) \times (\text{a} \to \text{b}) \to (\text{a} \to \text{c})\), but it doesn’t matter which type is used—the two types are equivalent, because the second is a consistent instantiation of the first (with new type variables).

As a cute and convenient library function, the ML library defines the infix operator \( o \) (that’s small-case character o, as in “oh my”) as function composition, just like in math. So instead of writing:

```plaintext
fun sqrt_of_abs i = Math.sqrt(Real.fromInt (abs i))
```

you could write:

```plaintext
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
```

But this second version makes clearer that we can just use function composition to create a function that we bind to a variable with a val-binding, as in this third version:

```plaintext
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

Although all three versions are fairly readable, the third one immediately communicates to the reader that \( \text{sqrt\_of\_abs} \) is just the composition of other functions.

**The pipeline operator.** Although the third version of \( \text{sqrt\_of\_abs} \) is concise, it—like function composition in mathematics—has the strange-to-many-programmers property that the computation proceeds from right-to-left. Programmers generally prefer to think left-to-right: “Take the absolute value, convert it to a real, and compute the square root.” But the mathematical version instead reads, “Take the square root of the conversion to real of the absolute value.”

So let’s define convenient syntax for left-to-right function composition. First, we define our own infix operator that lets us put the function to the right of the argument we are calling it with:

```plaintext
infix |> 
fun x |> f = f x
```

Note that \( |> \) here is the name of a function. Like the function named +, it is written as an infix operator. Now we can write:

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

Operator \( |> \) is commonly called the **pipeline operator**. It’s very popular in F# programming. As we’ve now seen, it’s purely syntactic sugar—there’s nothing complicated (or even interesting) about its semantics.
2 Fold

We previously covered two well-known and massively useful higher-order functions known as map and filter. Beyond those, a third incredibly useful higher-order function is fold, a.k.a. reduce and inject.

Sum and concatenation of lists. Let’s get to fold by way of a couple examples.

First, suppose we want to write a function to sum a list of integers. It should, by now, be quick work for you to write the following code:

```plaintext
fun sum l =
    case l of
        [] => 0
    | h::t => h + sum t
```

Second, suppose we want to concatenate a list of strings. You quickly write the following code:

```plaintext
fun concat l =
    case l of
        [] => ""
    | h::t => h ^ concat t
```

But now you notice something: those two functions are almost identical. In both functions, the code walks down a list, performing some operation with data at each step. In fact, the only differences are (1) the function name, (2) the base case for the recursion (0 vs. ""), and (3) the operator in the recursive case (+ vs. `).

Next, you decide that these are functions whose performance matters, so you want to make them tail recursive. That’s easy, too: you just apply the usual methodology we learned in class. You write the following code:

```plaintext
fun sum_tr l =
    let
        fun f(acc, l) =
            case l of
                [] => acc
            | h::t => f(h+acc, t)
    in
    f(0, l)
end

fun concat_tr l =
    let
        fun f(acc, l) =
            case l of
                [] => acc
            | h::t => f(h^acc, t)
    in
    f("", l)
end
```

And now you really start to suspect something interesting is happening, because helper function `f` in both cases is identical except for a single character!
When you notice two pieces of code that are doing almost exactly the same thing, you should think: is there some easier way to code this so that I only have to write the similar-looking code once? In the particular case of sum and concat, you should think: is there a way to abstract away from the details of traversing a list and doing something with each element?

Factoring out the list traversal. Looking at sum_tr and concat_tr, it becomes clear that there are only two real differences: (1) the initial value of the accumulator (0 vs. ""), and (2) the function that says what to “do” with each element in order to combine it with the accumulator (+ vs. ``). So let’s rewrite our helper functions f so that they take that “combiner” as an argument:

fun sum_tr l = 
  let
    fun f(combiner, acc, l) =
      case l of
        [] => acc
      | h::t => f(combiner, combiner(h,acc), t)
  in
    f(fn(x,y)=>x+y, 0, l)
  end

fun concat_tr l = 
  let
    fun f(combiner, acc, l) =
      case l of
        [] => acc
      | h::t => f(combiner, combiner(h,acc), t)
  in
    f(fn(x,y)=>y^x, "", l)
  end

Note that instead of fn (x,y) => x+y, for better style we should simply write op +, since that function already exists. But we can’t write op ^ instead of fn (x,y) => y^x, because of a subtle difference between the two: the latter puts the second string passed to it first.

Now the helper function f is actually identical in both cases. So let's factor out that code:

fun f(combiner, acc, l) =
  case l of
    [] => acc
  | h::t => f(combiner, combiner(h,acc), t)

fun sum_tr l = f(op +, 0, l)

fun concat_tr l = f(fn(x,y)="^x", "", l)

Folding from the left: foldl. This function f that we’ve discovered is actually an implementation of the well-known function fold. The primary purpose of fold is to iterate over a list, using a helper function combiner to combine all the elements of the list in some way. The particular implementation we discovered is called foldl, where the l means “left”:

fun foldl(f, acc, l) =
  case l of
foldl(f,acc,[x1,x2,...xn]) = f(xn...f(x2, f(x1, acc))...)

Why is it called “fold”? Because it takes an “initial answer” acc and uses f to “combine” acc and the first element of the list, using the result as the new “initial answer” for folding over the rest of the list. Clearly foldl is a higher-order function: it takes in a function f as an argument.

**Folding from the right: foldr.** Naturally, we can fold from the right:

foldr(f,acc,[x1,x2,...,xn]) = f(x1, f(x2, ...f(xn,acc)...))

However, foldr isn't tail recursive, so its performance won't be as good as foldl. It can be useful though, as seen in this example:

fun concat l = foldr(op ^, "", l)

Here, we actually get to use op ^, which is a more natural way to implement the concatenation of strings. (Compare to our implementation using foldl above.) If we wanted the best of both worlds, we could first reverse the list to make it possible to use op ^, then use foldl for performance:

fun concat_tr l = foldl(op ^, "", rev(l))

Here, rev is a function that reverses the list passed to it. Such a function exists in the standard library, but it's quite easy to code up yourself:

fun rev l =
  let
    fun f(acc, l) =
      case l of
        [] => acc
        | h::t => f(h::acc,t)
    in
      f([], l)
  end
Using fold to implement other higher-order functions. If you look closely at the implementation of \texttt{rev}, you’ll see something remarkable: it too is a tail-recursive traversal of a list, combining elements at each step with \texttt{cons}! So we could implement \texttt{rev} quite simply with \texttt{foldl}:

\begin{verbatim}
  fun rev l = foldl(op ::, [], l)
\end{verbatim}

Folding is so powerful, we can write several other functions using it:

\begin{verbatim}
  fun length l = foldl (fn (_,a) => a+1, 0, l)
  fun map(f,l) = foldr (fn (x,a) => (f x)::a, [], l)
  fun filter(f,l) = foldr (fn(x,a) => if f x then x::a else a, [], l)
\end{verbatim}

Using fold with combiners that have local information. Folding can become even more powerful when passing in helper functions that use in-scope variables to help them compute. 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.

\begin{verbatim}
  fun num_in_range (xs,lo,hi) = 
      fold ((fn (n,x) => n + (if lo <= x andalso x <= hi then 1 else 0)),
             0, xs)
  fun all_shorter (xs,s) = 
    let
      val i = String.size s
    in
      fold((fn (b,x) => b andalso String.size x < i), true, xs)
    end
\end{verbatim}

This pattern of separating the code doing the recursive traversal (\texttt{fold} or \texttt{map}) from the code doing the data-processing done on the elements (the anonymous functions in our examples) is fundamental. It is good to have this separation of concerns, because it leads to better programs.

3 Currying

The next idiom we consider is very convenient in general, and is often used when defining and using higher-order functions like map, filter, and fold. We have already seen that in ML 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. This technique is named currying in honor of a logician, Haskell Curry (1900–1982).

Defining and using curried functions Here is an example of a “three argument” function that uses currying:

\begin{verbatim}
  val sorted3 = fn x => fn y => fn z => z >= y andalso y >= x
\end{verbatim}

If we call \texttt{sorted3 4} we will get a function that has \texttt{x} in its environment. If we then call this function with 5, we will get a function that has \texttt{x} and \texttt{y} in its environment. If we then call this function with 6, we will get \texttt{true} because 6 is greater than 5 and 5 is greater than 4. So \texttt{((sorted3 4) 5) 6} computes exactly what we want and feels pretty close to calling \texttt{sorted3} with 3 arguments.
Even better, the parentheses are 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:

```ml
fun sorted3_tupled (x,y,z) = z >= y andalso y >= x
val 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.

As for defining a curried function, there's even better syntax available for it. We can actually write:

```ml
fun sorted3 x y z = z >= y andalso y >= x
```

With this syntactic sugar, you can just separate the conceptual arguments by spaces rather than using anonymous functions.

**Partial application.** Even though we might expect most clients of our curried `sorted3` to provide all 3 conceptual arguments, they might provide fewer and use the resulting function 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.

Currying is particularly convenient for creating functions that iterate over lists. For example, here is a curried version of `foldl`:

```ml
fun foldl f acc l =  
    case l of  
     [] => acc  
   | h::t => foldl f (f(h,acc)) t
```

Now we could use this fold to define a function that sums a list:

```ml
fun sum l = foldl (op+) 0 l
```

But that is unnecessarily complicated compared to just using partial application:

```ml
val sum = fold (op+) 0
```

The convenience of partial application is why many iterators in ML’s standard library use currying with the function they take as the first argument. For example, the types of all these functions use currying:

```ml
val List.map = fn : (‘a -> ‘b) -> ‘a list -> ‘b list
val List.filter = fn : (‘a -> bool) -> ‘a list -> ‘a list
val List.foldl = fn : (‘a * ‘b -> ‘b) -> ‘b -> ‘a list -> ‘b
```

As an example, `List.foldl((fun (x,y) => x+y), 0, [3,4,5])` does not type-check because `List.foldl` expects a `'a * 'b -> 'b` function, not a triple. The correct call is `List.foldl (fn (x,y) => x+y) 0 [3,4,5]`, which calls `List.foldl` with a function, which returns a function and so on.

**Currying without higher-order functions**

Although currying and partial application are great for higher-order functions, they are great even when working with functions that aren’t higher-order.
fun zip xs ys = 
    case (xs, ys) of 
      ([],[]) => []
    | (x::xs', y::ys') => (x,y) :: (zip xs' ys')
    | _ => raise Empty

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

val countup = range 1

fun pair_with_index xs = zip (countup (length xs)) xs

In this example, both \texttt{zip} and \texttt{range} are defined with currying, and \texttt{countup} partially applies \texttt{range}. The \texttt{pair_with_index} function turns the list \([v_1,v_2,...,v_n]\) into \([(1,v_1),(2,v_2),..., (n,v_n)]\).

\textbf{Curry and uncurl.} 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 our earlier idiom of combining functions can take functions using one approach and produce functions using another:

\begin{verbatim}
fun other_curry f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
\end{verbatim}

Looking at the types of these functions can help you understand what they do. As an aside, the types are also fascinating because if you pronounce \texttt{->} as “implies” and \texttt{*} as “and”, the types of all these functions are logical tautologies.

\section{The value restriction}

Once you have learned currying and partial application, you might try to use it to create a polymorphic function. Unfortunately, uses like this do not work in ML:

\begin{verbatim}
val mapSome = List.map SOME (*turn \[v_1,v_2,...,v_n]\ into \[\text{SOME} v_1, \text{SOME} v_2, ..., \text{SOME} v_n]\*)
val pairIt = List.map (fn x => (x,x)) (*turn \[v_1,v_2,...,v_n]\ into \[(v_1,v_1),(v_2,v_2),..., (v_n,v_n)]\*)
\end{verbatim}

Given what we have learned so far, there is no reason why this should not work, especially since all these functions do work:

\begin{verbatim}
fun mapSome xs = List.map SOME xs
val mapSome = fn xs => List.map SOME xs
val pairIt : int list -> (int * int) list = List.map (fn x => (x,x))
val incrementIt = List.map (fn x => x+1)
\end{verbatim}

The reason is called the \textit{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, but the type-checker does not know that.

The simplest approach is to ignore this issue until you get a warning/error about the value restriction. When you do, turn the val-binding back into a fun-binding like in the first example above of what works.