Functions as First-class Values

What does it mean for functions to be **first-class** data objects?

- They can be **arguments** to other functions.
- They can be **components** of data structures, such as tuples.
- They can be **returned** by other functions.

In practice, it's also desirable to have an **expression** syntax for describing them anonymously (the `fn` syntax).

From another point of view: a function type ($t_1 \rightarrow t_2$) should be valid wherever any other kind of type is valid, e.g.,

- $(t_1 \rightarrow t_2) \rightarrow t_3$
- $(t_1 \rightarrow t_2) \times (t_3 \rightarrow t_4)$
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$

The most distinctive thing about function values is that they cannot be **printed**; the only useful thing you can do with a function value (besides move it around) is to **apply** it.

Examples of Composition

Although the definition of composition requires an argument (`x`), we can **use** composition without thinking about arguments at all.

```
- fun f x = x + 1
val f = fn : int -> int
- fun g x = x + 2
val g = fn : int -> int
- val h = f o g
val h = fn : int -> int  (* h x = x + 3 *)

- val id : int -> int = (op~) o (op~);
val id = fn : int -> int

- fun millionaire worth = worth >= 1000000
val millionaire = fn : int -> bool
- fun g true = "yes"
    | g false = "no"
- val rich_enough = g o millionaire
val rich_enough = fn : int -> string

- fun f l = map (fn _ => 1,l)
val f = fn : 'a list -> int list
- fun sum l = reduce (op +,0,l)
val sum = fn : int list -> int
- val length = sum o f
val length = fn : 'a list -> int
```

Composition

In mathematics, the most familiar notion of a function that "returns another function" is the **composition** operator. It actually takes **two** functions f and g as arguments, and returns a function h of one argument x that behaves as follows:

- it applies g to x and gets back a result y
- it applies f to y and gets back another result z
- it returns z

We can define it in ML thus:

```
- fun compose(f,g) =
        let fun h x =
                let val y = g x
                in let val z = f y
                   in z
                   end
                end
        in h
        end
val compose = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Or, more compactly, and mimicing the standard infix name:

```
- fun f o g = fn x => f (g x)
val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
```

Partial Applications

An important source of functions that return functions is obtained by thinking about feeding **multi-argument** functions their arguments "one at a time."

For example, most arithmetic operators are binary, but we can build functions that fix one of the arguments, e.g.,

```
- fun addone x = 1 + x
val addone = fn : int -> int
- fun addtwo x = 2 + x
val addtwo = fn : int -> int
- map (addtwo,[1,2,3])
val it = [3,4,5] : int list
```

The payoff: we can use the resulting "partially applied" `+` function in any context where an `int -> int` function is needed.

Currying

Can we abstract over the process of "adding a fixed integer" to something? Yes!

```
- fun addint (y:int) =
   let fun f x = y + x
   in f
   end
val addint = fn : int -> int -> int

- (addint 1) 2;
val it = 3 : int
- addint 1 2;
val it = 3 : int
- val addone = addint 1
val addone = fn : int -> int
- addone 10;
val it = 11 : int
- map (addint 4,[3,4,5]);
val it = [7,8,9] : int list
```

Remember that the arrow type constructor associates to the right, so `int -> int -> int` is just shorthand for `int -> (int -> int)`.

Curried List Functions

The standard definitions of `map`, `reduce`, etc. are also curried, allowing partial applications and neater declarations:

```
- fun map f =
  let fun g nil = nil
        | g (h::t) = (f h)::(g t)
  in g
  end
val map = fn : ('a -> 'b) -> 'a list -> 'b list

- val add1toall = map (addint 1)
val add1toall = fn : int list -> int list
- add1toall [1,2,3];
val it = [2,3,4] : int list

- fun reduce f a =
  let fun g nil = a
        | g (h::t) = f(h,g t)
  in g
  end;
val reduce = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- val sum = reduce op+ 0;
val sum = fn : int list -> int
- sum [1,2,3];
val it = 6 : int
```

Currying (continued)

We could also have written `addint` in any of the following ways:

```
fun addint (y:int) = fn x => y + x

val addint = fn (y:int) => fn x => y + x

fun addint (y:int) x = y + x
```

These notations make it clear that all we're doing here is to **separate** the arguments to `+`, so that they can be given at different times.

This is called a **curried** definition of addition (after H.B. Curry). We can obviously do the same thing for any binary function, and in fact any n-ary function.

It's quite handy to do this for the built-in arithmetic operators, for example. (In some functional languages, the built-in operators are all curried to start with.)