Curried Functions

In principle, all functions take one argument, but the argument may be a tuple.

However, it is also possible to define a function with more than one parameter and no parentheses called *Curried* form. It makes a subtle difference in the type of the function.

Example:

```ml
fun add(x, y) = x+y:int;
val add = fn : int * int → int

fun addc x y = x+y:int;
val addc = fn : int → int → int
```

- `add` takes a pair of integers (an `int * int`) and returns their integer sum.
- `addc` takes one integer `x` as argument and returns a function that takes an integer `y` and adds `x` to it.
  - Note `->` associates from the right, so the type is `int->(int->int)`.

Partial Instantiation

We can name and assign this "intermediate" function.

```ml
val add3 = addc 3;
val add3 = fn : int → int

add3(10);
val it = 13 : int
```

Polymorphism

ML restricts types of variables only because it has to.

- A function takes a parameter of a given type.
  - e.g., `ord(s)` forces `s` to be a string.
• An overloaded function (e.g., +, <) applies to a variable, which must then be declared.

• A equality operator, = or <>, applies to a variable, forcing it to be an equality type.

  Equality types are defined recursively:

Basis: Elementary types (int, etc.) are equality types.

Induction: Tuples or lists of equality types are equality types.

fun ins gt (x, nil) = [x]
|   ins gt (x, y::ys) =
   if gt(x, y) then
     y::ins gt (x, ys)
   else x::y::ys;
val ins = fn : ('a * 'a) -> 'a * 'a list -> 'a list

fun isort gt nil = nil
| isort gt (x::xs) =
  ins gt (x, (isort gt xs));
val isort = fn : ('a * 'a) -> 'a list -> 'a list

isort (op >>) [3,1,4,1,5,9,2,6];
val it = [1,1,2,3,4,5,6,9] : int list

• op converts an infix operator like > into an “ordinary” function that takes a pair of arguments.

  Conversion is necessary because gt is of that form.

  fun igt(x:int,y) = x > y;
val igt = fn : int * int -> bool
val iisort = isort igt;
val iisort = fn : int list -> int list
iisort([5,3,7]);
val it = [3,5,7] : int list

Higher-Order Functions

ML makes no restrictions on function types.

• If $T_1$ and $T_2$ are any types, then $T_1 \to T_2$ is also a legal type, representing functions with domain type $T_1$ and range type $T_2$.  

• Any function whose arguments include one or more function types is a higher-order function.

Map
Among the interesting higher-order functions is:

```ml
fun map F nil = nil
  | map F (x::xs) = F(x)::map F xs;
val map = fn ('a -> 'b) -> 'a list -> 'b list
```

• Applies function $F$ to each element of a list and returns the resulting list.

• A Curried version of $\text{map}$ on p. 102, EMLP.

```ml
fun ++ x = x+1;
val ++ = fn : int -> int

map ++ [1,2,3];
val it = [2,3,4] : int list
```

• Remember that names composed of the usual symbols are legal identifiers in ML.

• We can also use an anonymous function as the first argument of $\text{map}$.  

```ml
map (fn x => x+1) [1,2,3]
val it = [2,3,4] : int list
```

• Finally, we can bind the first argument to create a function that applies to lists.

```ml
val listSq = map(fn x => x*x : int);
val listSq = fn : int list -> int list

listSq([1,2,3,4,5]);
val it = [1,4,9,16,25] ; int list
```

Reduce
• Put a (typically associative) operator between all the elements of a list and evaluate the resulting expression.

  e.g.: $[1,2,3,4]$ with * as the operator becomes $1 * 2 * 3 * 4 = 24$.

• We'll modify from p. 104, EMLP by also allowing an initial value associated with the empty list, and by Currying partially.
fun reduce (F,g) nil = g
  | reduce (F,g) (x::xs) = F(x,(reduce (F,g) xs));
val reduce = fn : ('a * 'b → 'b) * 'b → 'a list → 'b
reduce (op *, 1) [2,3,4,5];
val it = 120 : int

• The value of this expression is
  \[2 \times (3 \times (4 \times (5 \times 1)))\]
val length = reduce (fn(x,y) => y+1, 0);
val length = fn : 'a list → int

length(["a","b","c"]);
val it = 3 : int