Generalizing by Parameterizing

```
- val fibs1to10 =
  let fun f n =
        if n = 10 then [(10,fib 10)]
        else (n,fib n)::(f (n+1))
  in f 1
  end
```

How can we generalize this example?

One obvious idea is to make the start and stop indices into **parameters**. We'll neaten up the function a little while we're at it:

```
- fun fibs_p_to_q (p,q) =
  let fun f n =
       if n > q then nil
       else (n,fib n)::(f (n+1))
  in f p
  end;
val fibs_p_to_q = fn : int * int -> (int * int) list
- val fibs1to10 = fibs_p_to_q (1,10);
val fibs1to10 =
  [(1,1),(2,2),(3,3),(4,5),(5,8),(6,13),
   (7,21),(8,34),(9,55),(10,89)]
  : (int * int) list
- fibs_p_to_q (10,1);
val it = [] : (int * int) list
```

Now, what if we wanted to do something similar, but using the `fact` function instead of the `fib` function?

Choosing a parameterization

We could also have chosen a **more general** parameterization that doesn't force the output to be a list of pairs:

```
- fun gfuncs_p_to_q (func,p,q) =
  let fun f n =
       if n > q then nil
       else (func n)::(f (n+1))
  in f p
  end;
val gfuncs_p_to_q =
       fn : (int -> 'a) * int * int -> 'a list
- gfuncs_p_to_q (fib,1,10);
val it = [1,2,3,5,8,13,21,34,55,89] : int list
```

Now we can get the effect of the original function by passing a more complicated function as parameter:

```
- fun pairfib n = (n,fib n);
val pairfib = fn : int -> int * int
- gfuncs_p_to_q (pairfib,1,10);
val it = [(1,1),(2,2),(3,3),(4,5),(5,8),
          (6,13),(7,21),(8,34),(9,55),(10,89)]
  : (int * int) list
```

Or use an anonymous function:

```
- gfuncs_p_to_q (fn n => (n,fib n), 1,10);
val it = [(1,1),(2,2),(3,3),(4,5),(5,8),
          (6,13),(7,21),(8,34),(9,55),(10,89)]
  : (int * int) list
```

Functions as Parameters

Solution: **abstract** with respect to the `fib` function.

```
- fun funcs_p_to_q (func,p,q) =
  let fun f n =
       if n > q then nil
       else (n,func n)::(f (n+1))
  in f p
  end;
val funcs_p_to_q =
       fn : (int -> 'a) * int * int -> (int * 'a) list

- funcs_p_to_q(fib,1,5);
val it = [(1,1),(2,2),(3,3),(4,5),(5,8)]
         : (int * int) list

- funcs_p_to_q(fact,1,5);
val it = [(1,1),(2,2),(3,6),(4,24),(5,120)]
         : (int * int) list

- fun fibs_p_to_q (p,q) = funcs_p_to_q(fib,p,q);
val fibs_p_to_q = fn : int * int -> (int * int) list
```

Note that the first argument of `funcs_p_to_q` is (automatically) given the polymorphic type `int->'a`, which is tied to the output type `(int *'a) list`. So we can also write something like this:

```
- fun half n = real n / 2.0;
val half = fn : int -> real
- funcs_p_to_q(half,1,5);
val it =
  [(1,0.5),(2,1.0),(3,1.5),(4,2.0),(5,2.5)]
                               : (int * real) list
```

Higher-order mapping functions

Higher-order functions are great for operating on entire lists "at once":

Function `map` applies a function to each member of a list and returns the resulting list.

```
- fun map (f,nil) = nil
  | map (f,h::t) = (f h)::(map (f,t))
val map = fn : ('a -> 'b) * 'a list -> 'b list
- map (fn x => x + 1,[1,2,3]);
val it = [2,3,4] : int list
- map (size,["abc","de","f"]);
val it = [3,2,1] : int list
```

A variant of `map` is a function that applies a binary function to corresponding elements of **two** lists:

```
- fun map2(f,nil,nil) = nil
    | map2(f,ah::at,bh::bt) = f(ah,bh) :: (map2(f,at,bt))
std_in:2.2-3.57 Warning: match nonexhaustive
          (f,nil,nil) => ...
          (f,ah :: at,bh :: bt) => ...

val map2 =
   fn : ('a * 'b -> 'c) * 'a list * 'b list -> 'c list

- map2(op+,[1,2,3],[10,20,30])
val it = [11,22,33] : int list
- map2(op+,[1],[2,3]);

uncaught Match exception std_in:2.2-3.57
```

Higher-order functions with predicates

Function `filter` applies a **predicate** (a function that returns a `bool`) to each member of a list and returns a list containing just the matching members.

```
- fun filter (p,nil) = nil
    | filter (p,h::t) =
        if p h then
          h::(filter(p,t))
        else filter(p,t)
val filter = fn : ('a -> bool) * 'a list -> 'a list
- filter (fn x => x > 5, [1,5,8,7,9,2,1,6]);
val it = [8,7,9,6] : int list
```

Function `forall` takes a predicate and a list and returns true iff the predicate is true of every member. Note the "short-circuiting." `exists` can be defined similarly.

```
- fun forall (p,nil) = true
    | forall (p,h::t) =
       (p h) andalso forall(p,t);
- forall (fn x => x > 5, [6,7,9,8,4,7]);
val it = false : bool
```

List Reduction

Recall these examples:

```
- fun sum l =
    let fun g nil = 0
          | g (h::t) = (op +)(h,g t)
    in g l
    end;
val sum = fn : int list -> int
- fun concat l =
    let fun g nil = ""
          | g (h::t) = (op ^)(h,g t)
    in g l
    end;
val concat = fn : string list -> string
```

We can now see how to generalize these functions to a common higher-order function.

Both operate over lists, working from the right (tail) end, calculating a result.

• At each list item, the result is calculated by applying a binary **combining function** `f` (here `+` or `^`) to the list item and the result previously calculated for the tail.

• A special **seed value** `s` (here `0` or `""`) is needed to start off the computation at the tail of the list.

List Reduction (Continued)

Abstracting over the binary function and the seed value, we get the following function:

```
- fun reduce (f,s,l) =
    let fun g nil = s
          | g (h::t) = f(h,g t)
    in g l
    end
val reduce = fn : ('a * 'b -> 'b) * 'b * 'a list -> 'b

- fun sum l = reduce(op +,0,l);
val sum = fn : int list -> int
- fun concat l = reduce(op ^,"",l);
val concat = fn : string list -> string
```

If the list has contents $[x_1, x_2, x_3, \ldots, x_n]$ then we are computing

$$f(x_1, f(x_2, f(x_3, \ldots f(x_n, s)\ldots)))$$

If we wrote the list using a prefix form of "cons" it would look like this:

```
val cons = op::
cons(x1, cons(x2, cons(x3, ..., cons (xn, nil)...)))
```

Note that we can view `reduce(f,s,l)` as the result of replacing `cons` with `f` and `nil` with `s`.

Name that reduce!

Notice that the type of `reduce` is actually more general than we needed for these examples: the computed value and the list elements do not need to be of the same type:

```
- fun f(x,a) = size x + a
val f = fn : string * int -> int
- fun sumsize l = reduce(f,0,l);
val sumsize = fn : string list -> int
- sumsize ["abc","de","f"];
val it = 6 : int
```

A surprising number of functions can be written as instances of `reduce`,e.g.:

```
- fun mystery1 l = reduce(fn(_,a) => a+1,0,l)
val mystery1 = fn : 'a list -> int

- fun mystery2 (f,l) = reduce(fn(x,a) => f x::a, nil, l)
val mystery2 = fn : ('a -> 'b) * 'a list -> 'b list

- fun mystery3 (l,m) = reduce(op ::,m,l)
val mystery3 = fn : 'a list * 'a list -> 'a list

- fun mystery4 (p,l) =
      reduce(fn (x,a) => p x andalso a,true,l);
val mystery4 = fn : ('a -> bool) * 'a list -> bool
```