Higher-order functions

A higher-order function (or functional) is a function that operates on other functions; e.g., it either takes a function as an argument, or yields a function as a result.

Higher-order functions are a key feature that distinguishes functional from imperative programming. They naturally lead to:

- Partial evaluation.
- General-purpose functionals.
- Infinite lists.

Functionals with numbers

1. Summation.

   \[ \text{sum } f \ i \ j = \sum_{n=i}^{j} f(n) \]

   ```haskell
   fun sum f i j
   = if i > j then 0.0
     else f(i) + sum f (i+1) j 
   ~> val sum = fn : (int -> real) -> int -> int -> real
   
   "p01"
   > val sum = fn : (int -> real) -> int -> int -> real
   
   What do the functions
   fn f => fn i => fn k => fn l => sum ( sum f i ) k l
   and
   ```

2. Iterated composition.

   \[ \text{iterate } f \ n \ x = f^n(x) \]

   ```haskell
   fun iterate fn x
   = if n > 0 then iterate f (n-1) (f x)
     else x 
   ~> val 'a iterate
   = fn : ('a -> 'a) -> int -> 'a -> 'a
   ```
Map

It is often useful to systematically transform a list into another one by mapping each element via a function as follows

\[
\text{map } f \left[ a_1, \ldots, a_n \right] = [f(a_1), \ldots, f(a_n)]
\]

where \( a_i \in A \) \( (i = 1, \ldots, n) \) and \( f : A \to B \).

\[
\text{fun map } f \ [\ ] = [\ ]
\]
\[
| \text{map } f \ (h::t) = (f \ h) :: \text{map } f \ t ;
\]

"p03"

\[
> \text{val ('a, 'b) map }
= \text{fn : ('a -> 'b) -> 'a list -> 'b list}
\]

Filter

This functional applies a predicate (= boolean-valued function) to a list, returning the list of all the elements satisfying the predicate in the original order; thus filtering those that do not.

\[
\text{fun filter } P \ [\ ]
= [\ ]
\]
\[
| \text{filter } P \ (h::t)
= \text{if } P \ h \ \text{then } h :: \text{filter } P \ t \\
| \text{else filter } P \ t ;
\]

"p04"

\[
> \text{val 'a filter }
= \text{fn : ('a -> bool) -> 'a list -> 'a list}
\]

Fold

When folding a list, we compute a single value by folding each new list element into the result so far, with an initial value provided by the caller.

For \( a_i \in A \) \( (i = 1, \ldots, n) \), \( f : A \times B \to B \), and \( b \in B \) we have

\[
\text{fold left } \quad \text{foldl } f \ b \ [a_1, \ldots, a_n] = f(a_n, \ldots f(a_1, b) \ldots)
\]

and

\[
\text{fold right } \quad \text{foldr } f \ b \ [a_1, \ldots, a_n] = f(a_1, \ldots f(a_n, b) \ldots)
\]

NB: \text{foldl} and \text{foldr} are built-in functionals; \text{foldl} is tail recursive but \text{foldr} is not.
fun foldl f b [] = b
  | foldl f b (h::t) = foldl f (f(h, b)) t
fun foldr f b [] = b
  | foldr f b (h::t) = f(h, foldr f b t)

Examples:

1. val addall = foldl op\+ 0 ;
   val multall = foldl op\* 1 ;

Further list functionals

fun takewhile P [] = []
  | takewhile P (h::t) = if P h then h::takewhile P t else []
fun dropwhile P l
  = if null l then []
  else if P (hd l) then dropwhile P (tl l) else l
**Example**

```haskell
fun find P l
  = case dropwhile ( fn x => not(P x) ) l of
    [] => NONE
    | (h::_) => SOME h;
```

> val 'a find
  = fn : ('a -> bool) -> 'a list -> 'a option

**Examples:**

```haskell
infix isin ;
fun x isin l = exists (fn y => y = x) l ;
fun disjoint l1 l2
  = all (fn x => all (fn y => x<>y) l2) l1 ;
```

> infix 0 isin
> val 'a isin = fn : 'a * 'a list -> bool
> val 'a disjoint
  = fn : 'a list -> 'a list -> bool
```

**Matrix multiplication**

```haskell
fun dotprod l1 l2
  = foldl oplus 0.0 ( map op* (ListPair.zip( l1 , l2 )) ) ;
fun matmult Rows1 Rows2
  = let
      val Cols2 = transp Rows2
      in
      map (fn row => map (dotprod row) Cols2) Rows1
  end ;
```

> val dotprod = fn : real list -> real list -> real
> val matmult = fn :
  real list list -> real list list -> real list list
Generating permutations
A combinatorial version of the factorial function:

```plaintext
fun permgen [ ] = [ [ ] ]
| permgen l
= let fun
    pickeach [ ] = []
| pickeach (h::t)
    = (h,t) :: map (fn (x,l) => (x,h::l)) (pickeach t);
in
List.concat
 ( map (fn (h,l) => map (fn l => h::l) (permgen l))
   (pickeach l) )
end;
"p11"
```

Simple substitution cipher

```plaintext
load"ListPair";
fun makedict s t
= ListPair.zip( explode s , explode t ) ;

> val makedict
  = fn : string -> string -> (char * char) list

fun lookup D x
= case List.find (fn (s,t) => s=x) D of
    SOME(_,y) => y
  | NONE => x ;

> val "'a lookup = fn : (''a * ''a) list -> ''a -> ''a
```

```plaintext
fun encode D
  = foldl op^ "" o map (str o (lookup D)) o explode ;
val decode
  = encode o map (fn (s,t) => (t,s)) ;

"p12"

> val encode
  = fn : (char * char) list -> string -> string
> val decode
  = fn : (char * char) list -> string -> string
```