∼ Lecture IV ∼

Keywords:

higher-order functions; list functionals.

References:

- [MLWP, Chapter 3]

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.

$$\boxed{\text{sum } f\ i\ j = \sum_{n=i}^{j} f(n)}$$

```
fun sum f i j
  = if i > j then 0.0
    else f(i) + sum f (i+1) j ;
~
"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


```
fn h => fn i => fn j => fn k => fn l =>
  sum (fn n => sum (h n) i j) k l
```

do?

2. Iterated composition.

$$\boxed{\text{iterate } f\ n\ x = f^n(x)}$$

```
fun iterate f n x
  = if n > 0 then iterate f (n-1) (f x)
    else x ;
~
"p02"
> val 'a iterate
    = fn : ('a -> 'a) -> int -> 'a -> 'a
```

Map

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

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

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

```
fun map f [] = []
  | map f (h::t) = (f h) :: map f t ;
~
"p03"

> val ('a, 'b) map
    = fn : ('a -> 'b) -> 'a list -> 'b list
```


Examples

```
load"Real" ;
fun cast l = map (fn x => real x) l ;
fun scaleby n = map ( fn x => Real.*(n,x) ) ;
fun lift l = map (fn x => SOME x) l ;
fun transp ( []::_ ) = []
  | transp rows = (map hd rows) :: transp( map tl rows) ;
~
"p04"

> val scaleby = fn : int -> int list -> int list
> val 'a lift = fn : 'a list -> 'a option list
> val 'a matrixtransp = fn : 'a list list -> 'a list 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 **filter**ing those that do not.

```
fun filter P []
      = []
  | filter P (h::t)
      = if P h then h :: filter P t
        else filter P t ;

> val 'a filter
    = fn : ('a -> bool) -> 'a list -> 'a list
```


Fold

When **fold**ing 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

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

and

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

NB: `foldl` and `foldr` are built-in functionals; `foldl` is tail recursive but `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 ) ;
~
"p06"

> val ('d, 'e) foldl
    = fn : ('d * 'e -> 'e) -> 'e -> 'd list -> 'e
> val ('d, 'e) foldr
    = fn : ('d * 'e -> 'e) -> 'e -> 'd list -> 'e
```


Examples:

1.
```
val addall = foldl op+ 0 ;
val multall = foldl op* 1 ;
~
"p07"

> val addall = fn : int list -> int
> val multall = fn : int list -> int
```


2.
```
fun reverse l = foldl op:: [] l ;
fun length l = foldl ( fn (_,n)=> n+1 ) 0 l ;
fun append l = foldr op:: l ;
fun concat l = foldr op@ [] l ;
fun map f = foldr ( fn (h,t) => (f h)::t ) [] ;
~
"p07"

> val 'a reverse = fn : 'a list -> 'a list
> val 'a length = fn : 'a list -> int
> val 'a append = fn : 'a list -> 'a list -> 'a list
> val 'a concat = fn : 'a list list -> 'a list
> val ('a, 'b) map
    = fn : ('a -> 'b) -> 'a list -> 'b list
```


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 ;
~
"p08"

> val 'a takewhile
    = fn : ('a -> bool) -> 'a list -> 'a list
> val 'a dropwhile
    = fn : ('a -> bool) -> 'a list -> 'a list
```

Example

```
fun find P l
  = case dropwhile ( fn x => not(P x) ) l of
        [] => NONE
      | (h::_) => SOME h ;
~
"p08"
> val 'a find
    = fn : ('a -> bool) -> 'a list -> 'a option
```



```
fun exists P [] = false
  | exists P (h::t) = (P h) orelse exists P t ;
fun all P [] = true
  | all P (h::t) = (P h) andalso all P t ;
~
"p09"
> val 'a exists = fn : ('a -> bool) -> 'a list -> bool
> val 'a all = fn : ('a -> bool) -> 'a list -> bool
```


Examples:

```
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 ;
~
"p09"
> infix 0 isin
> val ''a isin = fn : ''a * ''a list -> bool
> val ''a disjoint
    = fn : ''a list -> ''a list -> bool
```


Matrix multiplication

```
fun dotprod l1 l2
  = foldl op+ 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 ;
~
"p10"
> 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:

```
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

```
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
```



```
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
```