Currying & Higher-Order Functions

fun plus (x : int) : int -> int = fn (y:int) => x + y

plus : int -> (int -> int)
    int -> int -> int   implicit right-associativity of parentheses for type arrows

basically syntactic sugar for
    val plus : int -> int -> int = fn (x : int) => fn (y:int) => x+y
    int -> int -> int

plus 3 : int -> int
a function value [3/x] fn (y: int) => x+y ≅ fn (y : int) => 3+ y

(plus 3) 4 : int ↪ 7

line up

Could have written plus 3 4   function application is left associative

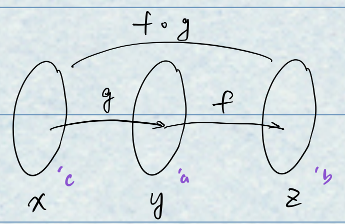
val p3 = plus 3
    p3 10 ↪ 13
    p3 2 ↪ 5

fun plus (x : int) (y : int) : int = x+y

val p4 = plus 4
    p4 10 ↪ 14
    plus 4 10 ↪ 14

fun plus x y = x + y   plus : int -> int -> int
    (turn * to ->)
fun add (x, y) = x + y   add : int * int -> int
always to supply pairs

plus is known as the curried version of add
    after Haskell Curry



$$(f \circ g)(x) = f(g(x))$$

fun o (f, g) = fn x => f(g(x))
infix o   op o : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

fun o (f, g) = f(g(x))   o is an example of a higher-order function
fun (f o g) x = f(g(x))

Order

ord (t) = 0 for base types

ord (t1 → t2) = max (ord (t1) + 1, ord (t2)) A function is higher order if its types has order at least 2

ord (int * int → int) = 1

ord (int → int → int) = 1 max (0 + 1, 1) = 1

ord 0 ord 1

```
fun inver x = x + 1        inver ∘ double ≅ fn x => 2x + 1
fun double x = 2 * x       double ∘ inver ≅ fn x => 2x + 2
```

```
(* filter : ('a -> bool) -> 'a list -> 'a list
   REQUIRES : p is total
   ENSURES : filter p l returns all elements in l for which the predicate returns true,
   in the original order
*)
fun filter p [] = []
  | filter p (x::xs) = (case p(x) of
                          true => x :: (filter p xs)
                        | false => filter p xs)
```

```
fun filter p =
  let
    fun f [] => []
      | f (x::xs) => (case p(x) of
                        true => x :: (f xs)
                      | false => f xs)
  in
    f
  end
```

no need to pass the recursive call

```
val keepevens = filter (fn n => (n mod 2) = 0)
```

int → bool

keepevens [2, 3, 4, 5, 6, 7] → [2, 4, 6]

filter (fn n => (n mod 2) = 0) [2, 3, 4, 5, 6, 7] → [2, 4, 6]

```
(* map : ('a -> 'b) -> 'a list -> 'b list
   map of [x1, ..., xn] ≅ [f(x1), ..., f(xn)]
*)
fun map _ [] = []
  | map f (x::xs) = (f x) :: (map f xs)
```

map Int.toString [1, 2, 3] → ["1", "2", "3"]

int → string int list string list

```
(* folder : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun folder f z [] = z
  | folder f z (x::xs) = f (x, folder f z x)
```

folder f z [x1, ..., xn] = f (x1, ... f (x_{n-1}, f (xn, z)))

(* foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldl f z [x1, ..., xn] = f(xn, ... f(x2, f(x1, z)))
*)

```
fun foldl _ z [] = z
  | foldl f z (x::xs) = foldl f (f(x, z)) xs
```

```
foldr (op +) 0 [1,2,3,4] ⇝ 10
foldl (op +) 0 [1,2,3,4] ⇝ 10
foldr (op -) 0 [1,2,3,4] ⇝ 1-(2-(3-(4-0))) = -2
foldl (op -) 0 [1,2,3,4] ⇝ 4-(3-(2-(1-0))) = 2
foldr (op ::) 0 [1,2,3,4] ⇝ [1,2,3,4]
foldl (op ::) 0 [1,2,3,4] ⇝ [4,3,2,1]
```

('b)
'a * 'a list -> 'a list

```
foldr (op ::) [] : 'a list -> 'a list
```
('a list)

≅ fn (L:'a list) => L

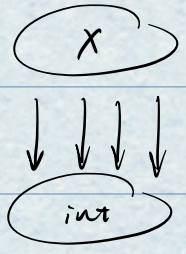
foldl (op ::) [] ≅ rev

Combinators

Informally, these are functions that combine "fragments of code into other fragments of code".

I think of them as hofs that expect functions & return a function. eg. o, map

There is a particular class of combinators that I like:



functions
X -> int

f : X -> int & g : X -> int

(f + g)(x) = f(x) + g(x) in math

pointwise principle

+, *, Int.min

In SML infix ++ (syntactic sugar)

```
fun (f ++ g) x = f(x) + g(x)
```

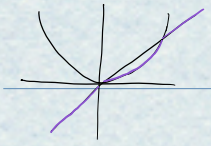
infixr **

```
fun (f ** g) x = f(x) * g(x)
```

```
fun MIN (f, g) x = Int.min (fx, gx)
```

fun f ++ g = fn x => f(x) + g(x)

```
fun square x = x * x
fun double x = 2 * x
val quadratic = square ++ double
```
≅ fn x => x ** x ++ 2 * x



```
val lower = MIN (square, double)
```

infix ++

```
fun (f ++ g) = f(x) + g(x)
```

(op ++) : ('a -> int) * ('a -> int) -> ('a -> int)

fun (f ++ g) = fn x => f(x) + g(x)

Staging

int * int -> int

```
fun f (x : int, y : int) : int =
```
takes 10 years

int -> int -> int

```
fun g (x : int) (y : int) : int =
```
takes 10 years

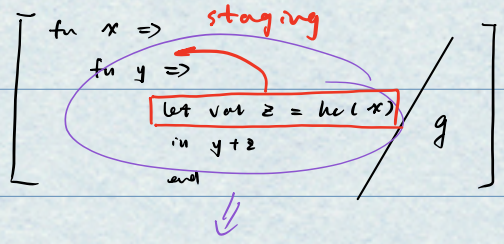
```
let
  val z : int = horrible computation (x)
in
  y + z
end
```

fun (5, 10) → 10 years

```
let
  val z : int = horrible computation (x)
in
  y + z
end
```

val g' = g 5 → takes 5 time

g' 10 → takes 10 yrs



this whole lamda express is a value, return instantly

int → int → int

```
fun h (x : int) : int → int =
  let
    val z : int = he(x)
  in
    fn y => y + z
  end
```

val h' = h 5 → takes 10 yrs

h' 10 → takes 5 time

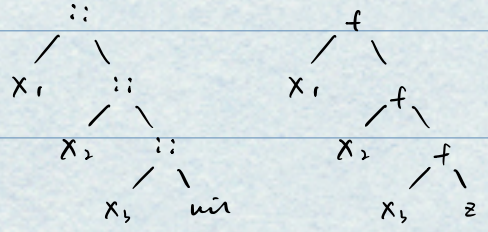
f —curry→ g —staged→ h

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* treemap : ('a → 'b) → 'a tree → 'b tree *)
fun treemap f map = Empty
  | treemap f (Node(l, x, r)) = Node (treemap f l, f x, treemap f r)

(* treefilter : ('a → bool) → 'a tree → 'a tree *)
fun treefilter p Empty = Empty
  | treefilter p (Node(l, x, r)) =
      if (p x) then Node (treefilter p l, x, treefilter p r)
      else combine (treefilter p l, treefilter p r)

fun combine (Empty, t2) = t2
  | combine (t1, Empty) = t1
  | combine (Node(l, x, r), t2) = Node (combine(l, t2), x, r)

val keepodds = treefilter (fn n => not (n mod 2 = 0)) : int tree → int tree
```

put every thing on the left

int → bool

eg. treeodds (3 / 4 7 / 1 12 2) => 3 / 1 7



Node (l, x, r) ~ f (_, _, _)

Empty ~ z

```
fun tfold f z Empty = z    ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b
  | tfold f z (Node(l,x,r)) = f(tfold f z l, x, tfold f z r)
```

```
SOME of _  ~  f        fun ofold f z NONE = z
NONE       ~  z          | ofold f z (SOME x) = f(x)
```

'a * 'a -> 'a Suppose this associative operation with a zero (g(x,z) ≅ x ≅ g(z,x))

```
fun gfold g z Empty = z
  | gfold g z (Node(l,x,r)) = g(gfold g z l, g(x, gfold g z r))
```

g(x, g(y, z)) ≅ g(g(x,y), z)