# Lec 10

**Recall**

fun add (x, y) = x + y

[ [ ](fn (x, y) ⇒ x + y) / add ]

**Consider**

fun plus x = fn y ⇒ x + y          plus : int → int → int

[ (fn x ⇒ fn y ⇒ x+y) / plus ]

(fn y ⇒ x+y → can be thought of as waiting for next arg)

val inc = plus 1          inc : int → int

[ [1/x] (fn y ⇒ x + y) / inc ]

inc 4 ⇒ [1/x] (fn y ⇒ x + y) 4
⇒ [1/x, 4/y] (x + y)
⇒ 1 + 4
⇒ 5

Or ... we can do one line?

plus 1 4 ⇒ [1/x] (fn y ⇒ x + y) 4

↳ Another way to do function call!
plus 1 4 ≅ add 1 4

Good compiler usually optimise this, so should be same as add

plus 1 4

"currying" — name from Haskell Curry

# Syntactic sugar for currying

fun plus x y = x+y

≅

fun plus x = (fn y => x + y)

≅

val plus = (fn x => (fn y => x+y))

fun sum x y z = x + y + z          sum : int → int → int → int

# Higher order function (HOF ...?)

Def: functions that takes in some function as arg or returns a HOF as output.

```
(* filter : ('a -> bool) -> 'a list -> 'a list
   REQ : p is total
   ENS : filter p L => L' s.t. L' ⊆ L and ∀l∈L, p(l), preserve order and multiplicity.  (sloppy)
*)
fun filter p [] = []
  | filter p x::xs = if p x then x:: filter p xs else filter p xs end
```

Note func app higher priority than infix ops

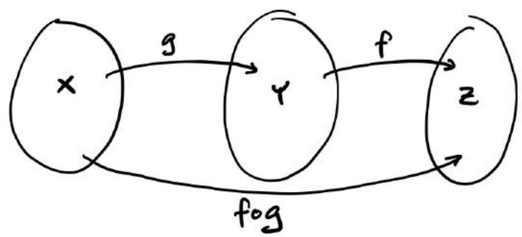
val keepevens = filter (fn n => n mod 2 = 0)   : int list → int list

keepevens [1,2,3,4,5] ↪ [2,4]

≅

filter (fn n => n mod 2 = 0) [1,2,3,4,5] ↪ [2,4]

# * Composing function




already built-in.

fun (f o g)(x) = f(g(x))

o : ('b -> 'c) * ('a -> 'b) -> 'a -> 'c

```
val increment = fn x = x+1
val double    = fn x = 2*x

increment o double : int -> int ≅ fn x => 2*x+1
double o increment : int -> int ≅ fn x => 2*x+2
```

# * The most famous HOF — "map"

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

map double [1,2,3] ↪ [2,4,6]

val doubler = map double    : int list -> int list
doubler [1,2,3] ↪ [2,4,6]
```

# Fold : accumulate result over whole list

foldl, foldr : ('a * 'b → 'b) → 'b → 'a list → 'b

Def: foldr f z [$x_1$, ..., $x_n$] = f($x_1$, ... f($x_{n-1}$, f($x_n$, z)) ... )

foldl f z [$x_1$, ..., $x_n$] = f($x_n$, ... f($x_2$, f($x_1$, z)) ... )

foldl (op +) 0 [1,2,3,4] ↪ 10
foldr (op +) 0 [1,2,3,4] ↪ 10

foldl (op -) 0 [1,2,3,4] ≅ 4-(3-(2-(1-0))) ↪ 2
foldr (op -) 0 [1,2,3,4] ≅ 1-(2-(3-(4-0))) ↪ -2

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

↳ updated accumulator
↳ tail recursive

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

↳ uh oh not tail recursive

ASIDE
foldr is "catamorphism" for list.
see cat theory (?)
rather typical in data strucs

---

foldr (op ::) [] ≅ id
foldl (op ::) [] ≅ rev
foldr (op ::) Y X ≅ X@Y
foldl (op ::) acc L ≅ rev L @ acc