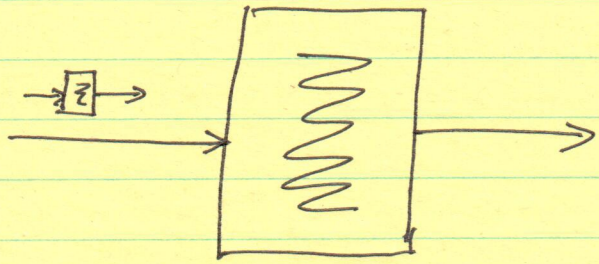
Higher-order functions:



A function can be input to another f^n.

map : ('a → 'b) → 'a list → 'b list
filter ('a → bool) → 'a list → 'a list

—x—

Increasing levels of abstraction:

```
let rec sumInts (a,b) =
    if a>b then 0
    else a + sumInts (a+1, b)

let rec sumSquares (a,b) =
    if a>b then 0
    else (a*a) + sumSquares (a+1, b)

let rec sumCubes (a, b) =
    if a>b then 0
    else (a*a*a) + sumCubes (a+1, b)
```

Aha! the square, cube etc. should be abstracted

```
let rec sum (f, lo, hi) =
    if lo > hi then 0
    else (f lo) + sum (f, lo+1, hi)
```

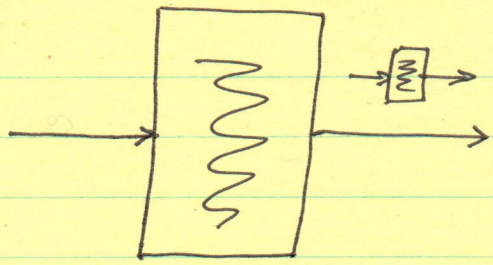
```
let rec sumInc (f, lo, hi, inc) =
      if (lo > hi) then 0
      else (f lo) + sumInc (f, (inc lo), hi, inc)
```

((int → int) * int * int * (int → int)) ⟶ int

```
let rec product (f, lo, hi, inc) =
      if (lo > hi) then 1
      else (f lo) * product (f, (inc lo), hi, inc)
```

```
let acc (comb, f, lo, hi, inc, unit) =
    let rec helper (a, tmp) =
        if (a > hi) then tmp
        else
            helper ((inc a), comb (tmp, (f a)))
    in
        helper (lo, unit)
```

EXERCISE : Please figure out the type of this function.



The output is also perhaps a function.
EXAMPLE : derivatives in calculus

$$\frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

$$\frac{f(x+dx) - f(x)}{dx}$$

```
let deriv(f, dx) =
    fun x -> ((f (x +. dx)) -. (f x)) /. dx
```

(float -> float) * float -> (float -> float)

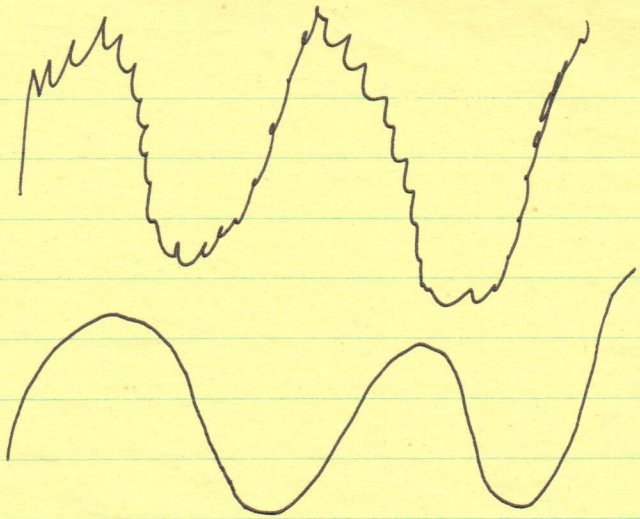
```
let iterSum (f, lo, hi, inc) =
    let rec helper(x, result) =
        if (x > hi) then result
        else helper ((inc x), (f x) +. result)
    in
    helper(lo, 0.0)

let integral (f, lo, hi, dx) =
    let delta x = x +. dx in
      dx *. iterSum(f, (lo +. (dx/.2)), hi, delta)
```

$$\int_{lo}^{hi} f\, dx \rightarrow \text{Definite integral}$$

How should one write an indefinite integral?

smoothing



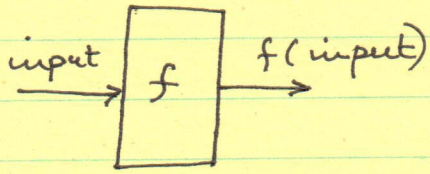
```
let smooth f delta =
    fun x ->
        (f (x -. delta)) +. (f x) +. (f (x +. delta)) /. 3.0
```

CURRYING (HASKELL CURRY)
[COMBINATORY LOGIC]

$$A * B \to C \simeq A \to (B \to C)$$

```
let myadd (m, n) = m + n
        int * int -> int
    myadd 3 -> type error
let myadd2 m n = m + n
    int -> int -> int
    myadd2 3 -> fun of type int -> int
```

Pipes



f input

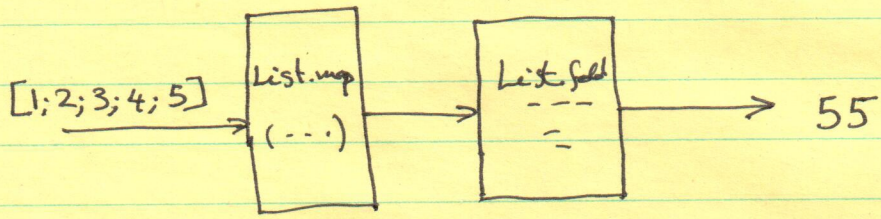
input |> f = f input

pipe

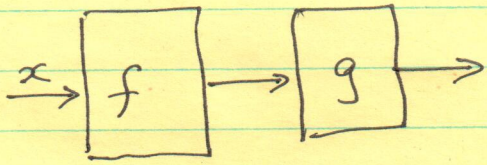
```
let inc n = n+1;;
List.map inc [1;2;3] ~~> [2;3;4]
```

```
[1;2;3] |> (List.map inc)
```

```
(1--10) |> (List.fold_left (+) 0) ~~> 55
(1--5) |> (List.map (fun n -> n*n)) |> (List.fold_left (+) 0)
```




```
let compose f g =
    fun x -> x |>f |>g
```




```
let (--) lo hi = List.init (hi-lo+1) (fun n -> n+lo)
```