Higher-Order Functions I

15-150

Lecture 10: September 30, 2025

Stephanie Balzer
Carnegie Mellon University

Recap

- Polymorphic types (type families) and instantiation.
  - parameterized data types

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

- Typing rules and most general type.
- Type guarantee:

$$\text{If } e : t \text{ and } e \hookrightarrow v \text{ then } v : t.$$

Today's topic

- Function currying.
- Higher-order functions:
  - Functions whose argument and/or return types are functions.
- More polymorphism.

Currying

Consider:

```
fun add (x,y) = x + y
```

What's its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add:                    *)
```

What's its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int        *)
```

What’s its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)
```

What's its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)
```

Recall, the function definition introduces a binding in the environment:

```
[ env                        ]
[ fn (x,y) => x + y   /add   ]
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)
```

Let’s consider another function:

```
fun plus x = fn y => x + y
```

What’s its type?

Curying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)
```

Let’s consider another function:

```
fun plus x = fn y => x + y
                                (* plus:                  *)
```

What’s its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)
```

Let’s consider another function:

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

What’s its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)
```

Let’s consider another function:

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

What’s its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)
```

Let’s consider another function:

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

What’s its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Notice, arrows right-associate!

t_1 -> t_2 -> t_3 means t_1 -> (t_2 -> t_3)

Correspondingly, function application left-associates!

`f x y` means `(f x) y`

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Binding for plus:

```
[ env
  fn x => fn y => x + y /plus]
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

- Function `plus` is the curried form of `add`.
- Currying: changing “*” to “->”.
    - Named after Haskell Curry.
    - Useful for staging (next lecture!).

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let's evaluate:

```
[…,(fn (x,y) => x+y)/add] add (3,4)
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let’s evaluate:

```
[…,(fn (x,y) => x+y)/add] add (3,4)
==> […] (fn (x,y) => x+y) (3,4)
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let’s evaluate:

```
[…,(fn (x,y) => x+y)/add] add (3,4)
==> […] (fn (x,y) => x+y) (3,4)
==> […, 3/x, 4/y] x+y
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let’s evaluate:

```
[…,(fn (x,y) => x+y)/add] add (3,4)
==> […] (fn (x,y) => x+y) (3,4)
==> […, 3/x, 4/y] x+y
==> […, 3/x, 4/y] 3+y
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let's evaluate:

```
[…,(fn (x,y) => x+y)/add] add (3,4)
==> […] (fn (x,y) => x+y) (3,4)
==> […, 3/x, 4/y] x+y
==> […, 3/x, 4/y] 3+y
==> […, 3/x, 4/y] 3+4
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let’s evaluate:

```
[…,(fn (x,y) => x+y)/add] add (3,4)
==> […] (fn (x,y) => x+y) (3,4)
==> […, 3/x, 4/y] x+y
==> […, 3/x, 4/y] 3+y
==> […, 3/x, 4/y] 3+4
==> […] 7
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let’s define yet another function:

```
val incr3 = plus 3
```

What’s its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let’s define yet another function:

```
val incr3 = plus 3      (* incr3:             *)
```

What’s its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let’s define yet another function:

```
val incr3 = plus 3        (* incr3: int ->     *)
```

What’s its type?

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

Let’s define yet another function:

```
val incr3 = plus 3         (* incr3: int -> int *)
```

What’s its type?

partial application

Currying

Consider:

```
fun add (x,y) = x + y    (* add: int * int -> int *)

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

val incr3 = plus 3       (* incr3: int -> int *)
```

Binding for incr3:

```
[ env
  [3/x]
  fn y => x + y  / incr3 ]
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

val incr3 = plus 3      (* incr3: int -> int *)
```

Let's evaluate:

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

val incr3 = plus 3      (* incr3: int -> int *)
```

Let's evaluate:

```
[…,(fn x => fn y => x+y)/plus] plus 3
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

val incr3 = plus 3          (* incr3: int -> int *)
```

Let's evaluate:

```
[…,(fn x => fn y => x+y)/plus] plus 3
==> […] (fn x => fn y => x+y) 3
```

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

val incr3 = plus 3         (* incr3: int -> int *)
```

Let's evaluate:

```
[…,(fn x => fn y => x+y)/plus] plus 3
==> […] (fn x => fn y => x+y) 3
==> […, 3/x] fn y => x+y
```

It's a value!

lambda

Currying

Consider:

```
fun add (x,y) = x + y   (* add: int * int -> int *)

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

val incr3 = plus 3         (* incr3: int -> int *)
```

Let's evaluate:

```
[…,(fn x => fn y => x+y)/plus] plus 3
==> […] (fn x => fn y => x+y) 3
==> […, 3/x] fn y => x+y
```

closure

Currying

Consider:

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

val incr3 = plus 3        (* incr3: int -> int *)
```

Currying

Consider:

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

val incr3 = plus 3        (* incr3: int -> int *)
```

Let’s evaluate:

```
[…, 3/x, (fn y => x+y)/incr3] incr3 4
```

Currying

Consider:

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

val incr3 = plus 3            (* incr3: int -> int *)
```

Let's evaluate:

```
[…, 3/x, (fn y => x+y)/incr3] incr3 4
==> […, 3/x] (fn y => x+y) 4
```

Currying

Consider:

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

val incr3 = plus 3        (* incr3: int -> int *)
```

Let's evaluate:

```
[…, 3/x, (fn y => x+y)/incr3] incr3 4
==> […, 3/x] (fn y => x+y) 4
==> […, 3/x, 4/y] x+y
```

Currying

Consider:

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

val incr3 = plus 3              (* incr3: int -> int *)
```

Let’s evaluate:

```
[…, 3/x, (fn y => x+y)/incr3] incr3 4
==> […, 3/x] (fn y => x+y) 4
==> […, 3/x, 4/y] x+y
==> […, 3/x, 4/y] 3+y
```

Currying

Consider:

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

val incr3 = plus 3                 (* incr3: int -> int *)
```

Let's evaluate:

```
[…, 3/x, (fn y => x+y)/incr3] incr3 4
==> […, 3/x] (fn y => x+y) 4
==> […, 3/x, 4/y] x+y
==> […, 3/x, 4/y] 3+y
==> […, 3/x, 4/y] 3+4
```

Currying

Consider:

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

val incr3 = plus 3                (* incr3: int -> int *)
```

Let's evaluate:

```
[…, 3/x, (fn y => x+y)/incr3] incr3 4
==> […, 3/x] (fn y => x+y) 4
==> […, 3/x, 4/y] x+y
==> […, 3/x, 4/y] 3+y
==> […, 3/x, 4/y] 3+4
==> […] 7
```

Currying: syntactic sugar

A maybe more convenient notation

```
fun plus x y = x + y
```

space!

which is syntactic sugar for

```
fun plus x = fn y => x + y
```

Currying: syntactic sugar

A maybe more convenient notation

```
fun plus x y = x + y
```

which is syntactic sugar for

```
fun plus x = fn y => x + y
```

which is itself syntactic sugar for

```
val rec plus = fn x => fn y => x + y
```

`fun` definitions are recursive!

Currying: some more examples

```
fun f (x, y, z) = (x + y) div z
(* f:                              *)
```

Currying: some more examples

```
fun f (x, y, z) = (x + y) div z
(* f: int * int * int -> int *)

fun g x y z = (x + y) div z
(* g:                          *)
```

Currying: some more examples

```
fun f (x, y, z) = (x + y) div z
(* f: int * int * int -> int *)

fun g x y z = (x + y) div z
(* g: int -> int -> int -> int *)

fun h (x, y) z = (x + y) div z
(* h:                          *)
```

Currying: some more examples

```
fun f (x, y, z) = (x + y) div z
(* f: int * int * int -> int *)

fun g x y z = (x + y) div z
(* g: int -> int -> int -> int *)

fun h (x, y) z = (x + y) div z
(* h: int * int -> int -> int *)
```

Higher-order functions

Higher-order functions are functions whose argument type and/or return type are functions.

Remark: Some require return type to be a higher-order function.
We do not adopt this stricter definition.

Higher-order functions facilitate writing parametric code.

Higher-order functions facilitate staging.

Higher-order function: filter

Filtering elements in a list, given a predicate:

```
(* filter: ('a -> bool) -> 'a list -> 'a list
   REQUIRES: p is total
   ENSURES: filter p L  evaluates to a list consisting of
            the elements of L that satisfy p,
            with elements appearing in same order as in L.
*)
```

- `filter` is higher-order
  - takes the predicate `p: ‘a -> bool` as an argument
- `filter` is curried
- `filter` is predefined as `List.filter`.

Higher-order function: filter

Filtering elements in a list, given a predicate:

```
(* filter: ('a -> bool) -> 'a list -> 'a list
   REQUIRES: p is total
   ENSURES: filter p L  evaluates to a list consisting of
            the elements of L that satisfy p,
            with elements appearing in same order as in L.
*)

fun filter (p: 'a -> bool) ([]: 'a list): 'a list =
  | filter p (x::xs) =
```

Higher-order function: filter

Filtering elements in a list, given a predicate:

```
(* filter: ('a -> bool) -> 'a list -> 'a list
   REQUIRES: p is total
   ENSURES: filter p L  evaluates to a list consisting of
            the elements of L that satisfy p,
            with elements appearing in same order as in L.
*)

fun filter (p: 'a -> bool) ([]: 'a list): 'a list = []
  | filter p (x::xs) =
```

Higher-order function: filter

Filtering elements in a list, given a predicate:

```
(* filter: ('a -> bool) -> 'a list -> 'a list
   REQUIRES: p is total
   ENSURES: filter p L  evaluates to a list consisting of
            the elements of L that satisfy p,
            with elements appearing in same order as in L.
*)

fun filter (p: 'a -> bool) ([]: 'a list): 'a list = []
  | filter p (x::xs) = if p(x) then
                                 else
```

Higher-order function: filter

Filtering elements in a list, given a predicate:

```
(* filter: ('a -> bool) -> 'a list -> 'a list
   REQUIRES: p is total
   ENSURES: filter p L  evaluates to a list consisting of
            the elements of L that satisfy p,
            with elements appearing in same order as in L.
*)

fun filter (p: 'a -> bool) ([]: 'a list): 'a list = []
  | filter p (x::xs) = if p(x) then x::(filter p xs)
                               else
```

Higher-order function: filter

Filtering elements in a list, given a predicate:

```
(* filter: ('a -> bool) -> 'a list -> 'a list
   REQUIRES: p is total
   ENSURES: filter p L  evaluates to a list consisting of
            the elements of L that satisfy p,
            with elements appearing in same order as in L.
*)

fun filter (p: 'a -> bool) ([]: 'a list): 'a list = []
  | filter p (x::xs) = if p(x) then x::(filter p xs)
                                else filter p xs
```

Higher-order function: filter

Filtering elements in a list, given a predicate:

```
fun filter (p: 'a -> bool) ([]: 'a list): 'a list = []
  | filter p (x::xs) = if p(x) then x::(filter p xs)
                                else filter p xs
```

Filter in action:

```
                         (* int -> bool *)
val keepevens = filter (fn n => n mod 2 = 0)
```

type?

Higher-order function: filter

Filtering elements in a list, given a predicate:

```
fun filter (p: 'a -> bool) ([]: 'a list): 'a list = []
  | filter p (x::xs) = if p(x) then x::(filter p xs)
                                else filter p xs
```

Filter in action:

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

Higher-order function: filter

Filtering elements in a list, given a predicate:

```
fun filter (p: 'a -> bool) ([]: 'a list): 'a list = []
  | filter p (x::xs) = if p(x) then x::(filter p xs)
                                else filter p xs
```

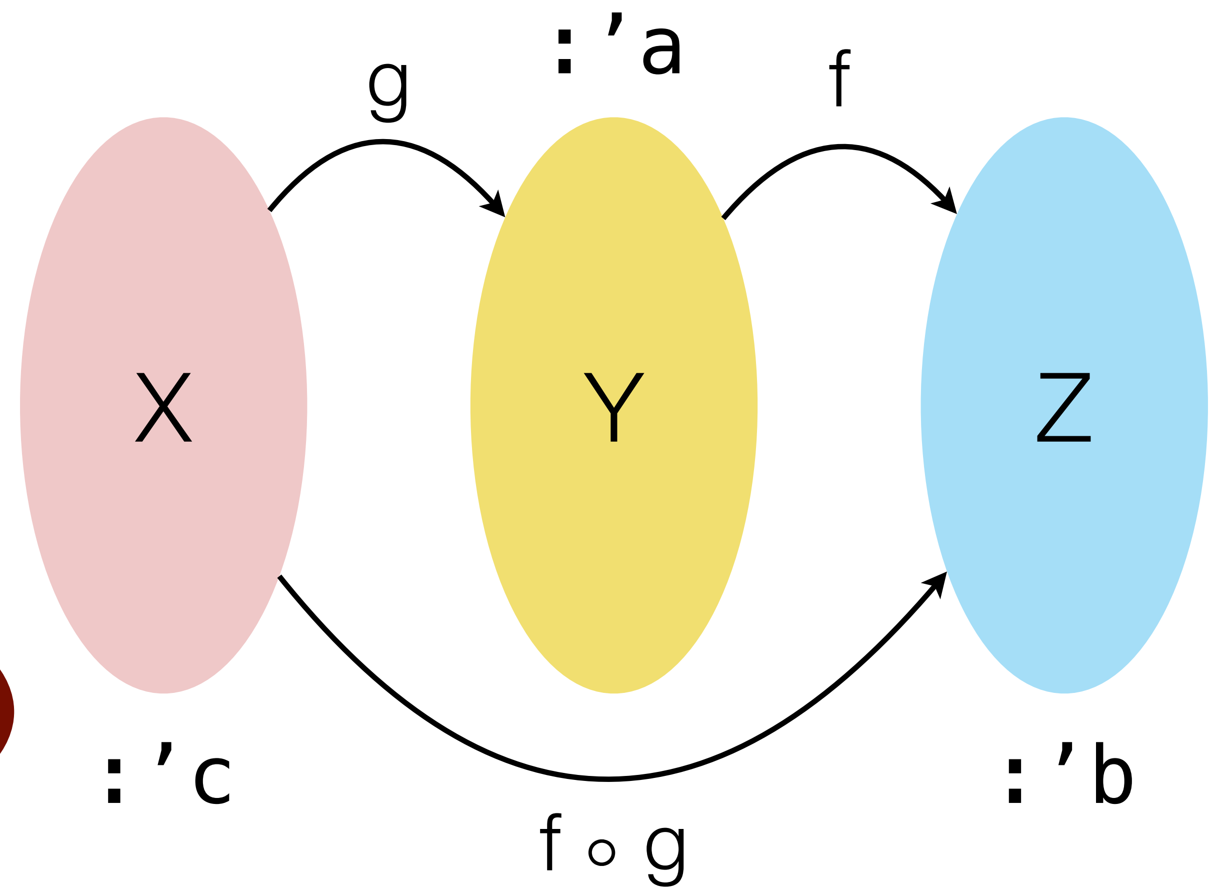
Filter in action:

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

val [2,~6, 10] = keepevens [1,2,~5,~6,11,10,13]
```

Higher-order function: composition

Function composition, abstractly: $(f \circ g)\ (x) = f\ (g\ (x))$



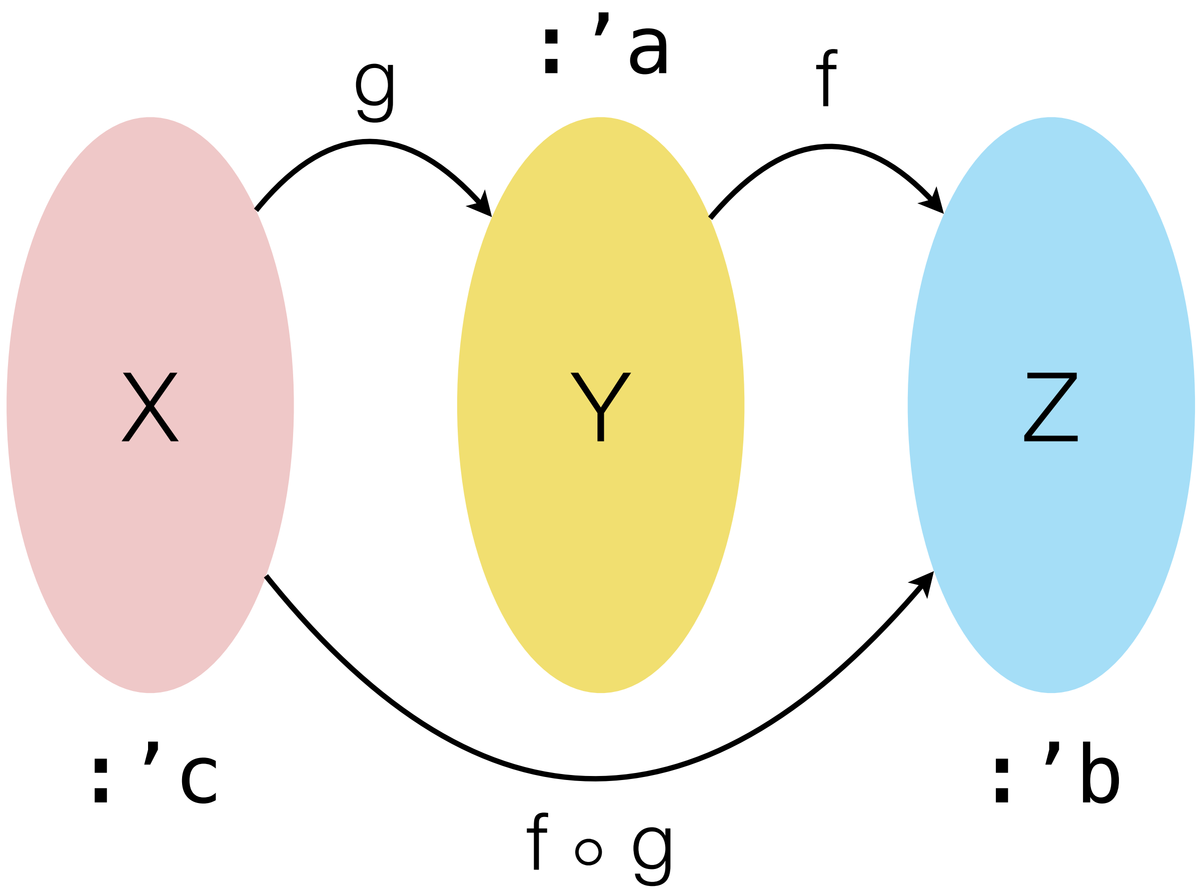
predefined in SML

∘ is a higher-order function, i.e., a combinator, expecting two functions as arguments and returning a function.

Higher-order function: composition

```
infix o
fun f o g = fn x => f(g(x))
```

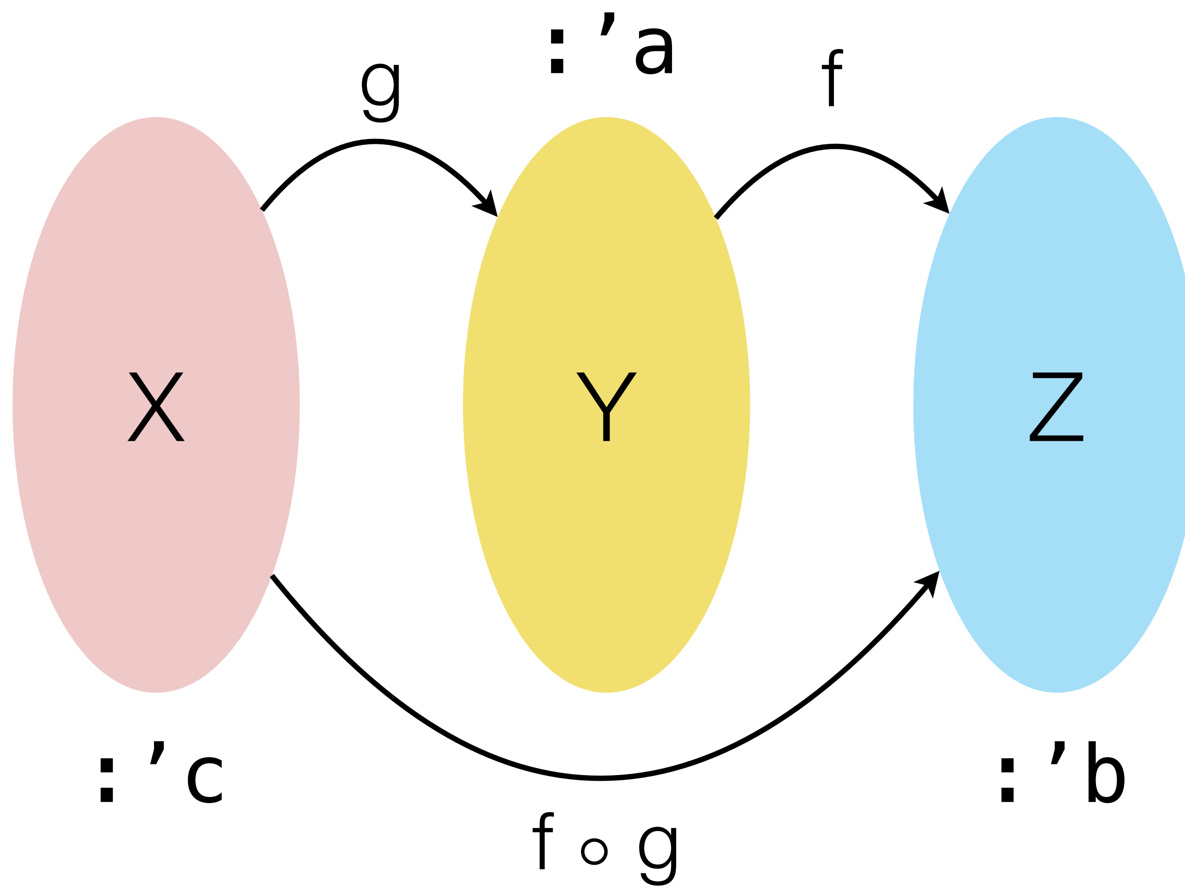
What is its type?



Higher-order function: composition

```
infix o
fun f o g = fn x => f(g(x))
```

What is its type?

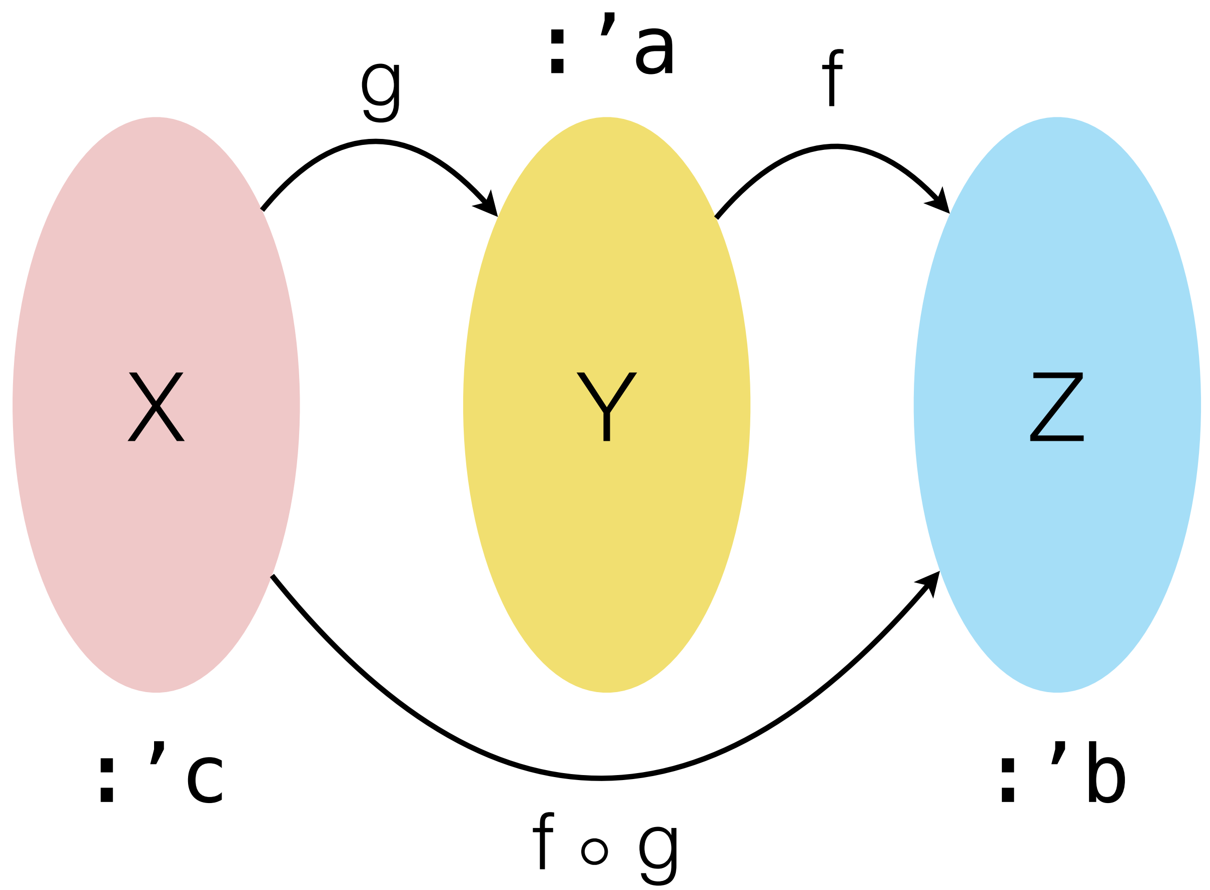



```
(* (op o):                                   *)
```

Higher-order function: composition

```
infix o
fun f o g = fn x => f(g(x))
```

What is its type?

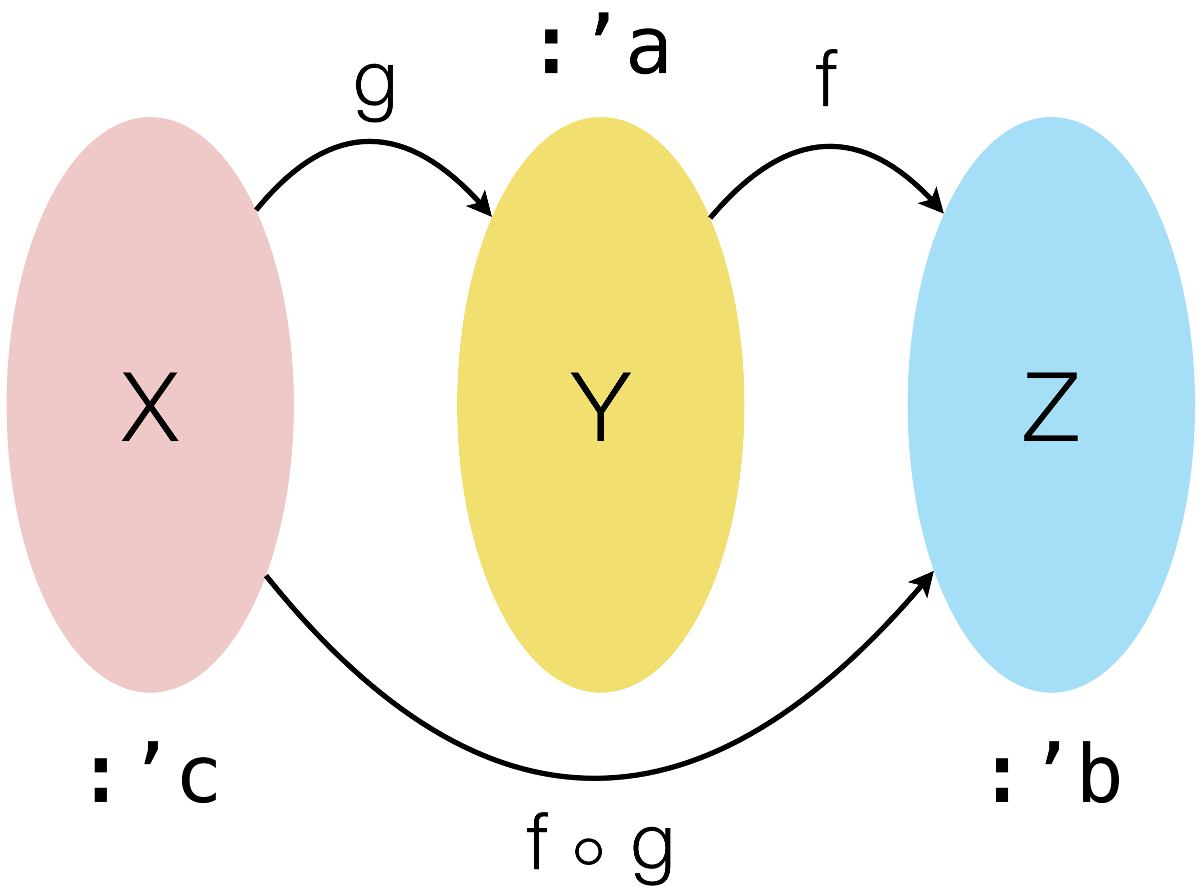



```
(* (op o): ('a -> 'b) *                          *)
```

Higher-order function: composition

```
infix o
fun f o g = fn x => f(g(x))
```

What is its type?

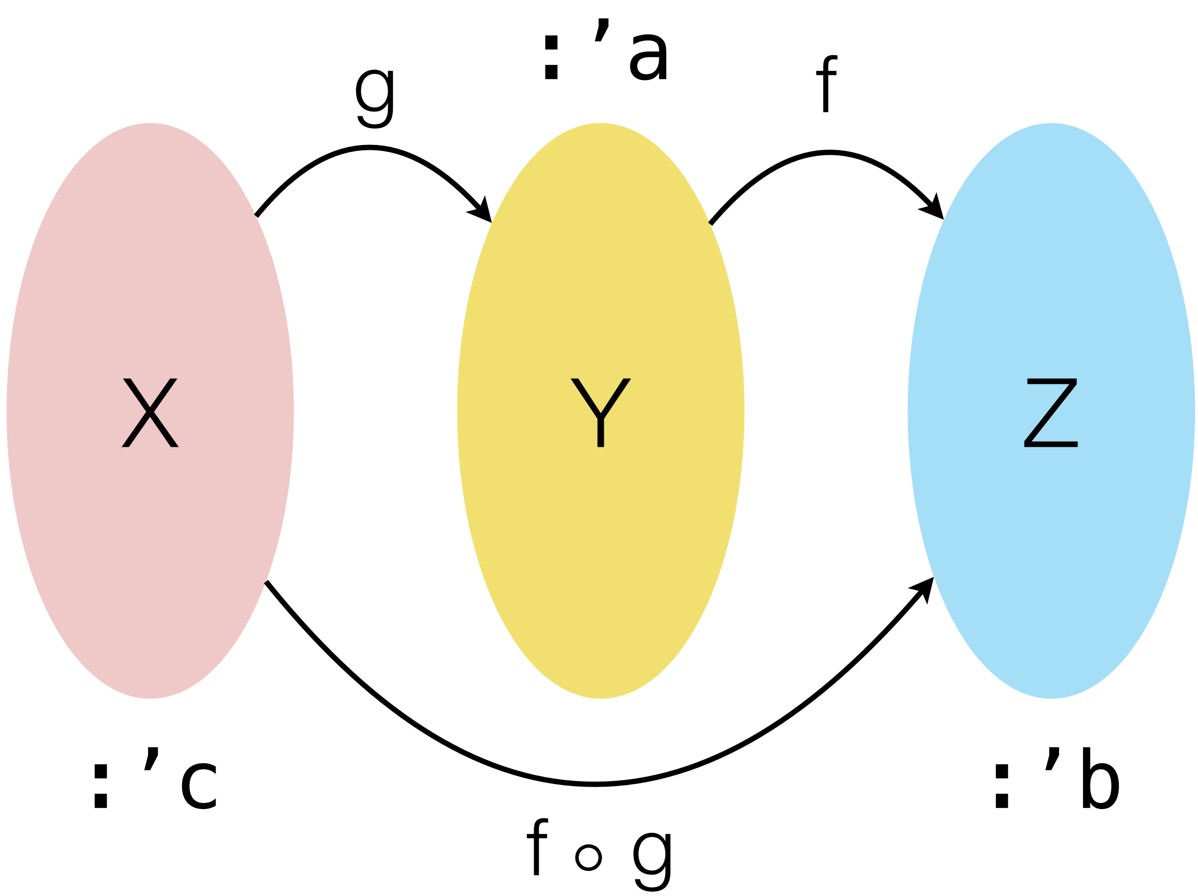



```
(* (op o): ('a -> 'b) * ('c -> 'a)                *)
```

Higher-order function: composition

```
infix o
fun f o g = fn x => f(g(x))
```

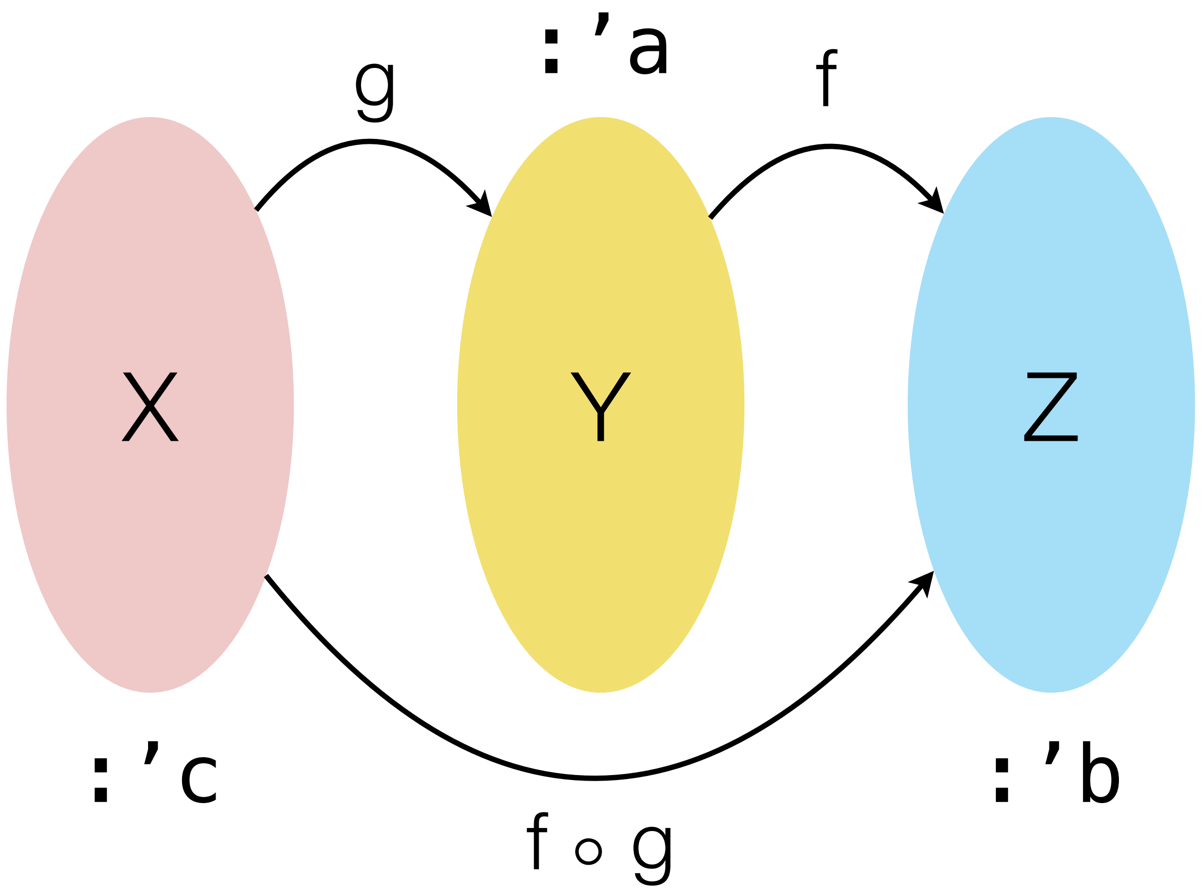
What is its type?




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

Higher-order function: composition

```
infix o
fun f o g = fn x => f(g(x))
```



Examples:

```
fun incr x = x + 1
fun double x = 2 * x
```

Then we have:

double o incr $\cong$ fn x => 2x + 2

incr o double $\cong$ fn x => 2x + 1

Higher-order function: map

Transforming elements in a list, given a transformation function:

```
(* map: ('a -> 'b) -> 'a list -> 'b list
   REQUIRES: true
   ENSURES: For all n≥0, map f [x1,…,xn] ≅ [f x1,…,f xn]
*)
```

- `map` is higher-order
  - takes a function `f: 'a -> 'b` as an argument
- `map` is predefined in SML

Higher-order function: map

Transforming elements in a list, given a transformation function:

```
(* map: ('a -> 'b) -> 'a list -> 'b list
   REQUIRES: true
   ENSURES: For all n≥0, map f [x1,…,xn] ≅ [f x1,…,f xn]
*)
```

```
fun map (f: 'a -> 'b) ([]: 'a list): 'b list =
  | map f (x::xs) =
```

Higher-order function: map

Transforming elements in a list, given a transformation function:

```
(* map: ('a -> 'b) -> 'a list -> 'b list
   REQUIRES: true
   ENSURES: For all n≥0, map f [x1,…,xn] ≅ [f x1,…,f xn]
*)

fun map (f: 'a -> 'b) ([]: 'a list): 'b list = []
  | map f (x::xs) =
```

Higher-order function: map

Transforming elements in a list, given a transformation function:

```
(* map: ('a -> 'b) -> 'a list -> 'b list
   REQUIRES: true
   ENSURES: For all n≥0, map f [x1,…,xn] ≅ [f x1,…,f xn]
*)
```

```
fun map (f: 'a -> 'b) ([]: 'a list): 'b list = []
  | map f (x::xs) = f(x)::(map f xs)
```

Example:

```
map double [1,2,3] ==> [2,4,6]
```

Higher-order function: fold

Combining elements in a list, given a binary operation and base value:

```
(* fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
```

combining function:

'a: type of list elements

'b: type of base value and of combined value

initial value

list to be combined

combined value

Higher-order function: fold

Combining elements in a list, given a binary operation and base value:

```
(* fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
```

Two implementations:

$$\texttt{foldl f z } [x_1,\ldots,x_n] \cong \texttt{f}(x_n,\ldots\texttt{f}(x_3,\texttt{f}(x_2,\texttt{f}(x_1,\texttt{z}))))$$

$$\texttt{foldr f z } [x_1,\ldots,x_n] \cong \texttt{f}(x_1,\ldots\texttt{f}(x_{n-2},\texttt{f}(x_{n-1},\texttt{f}(x_n,\texttt{z}))))$$

Higher-order function: fold

Combining elements in a list, given a binary operation and base value:

```
(* fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
```

Two implementations:

$$\texttt{foldl f z } [x_1,\dots,x_n] \cong f(x_n,\dots f(x_3,f(x_2,f(x_1,z))))$$

$$\texttt{foldr f z } [x_1,\dots,x_n] \cong f(x_1,\dots f(x_{n-2},f(x_{n-1},f(x_n,z))))$$

Examples:

```
foldl (op -) 0 [1,2,3,4] ==>
```

Higher-order function: fold

Combining elements in a list, given a binary operation and base value:

```
(* fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
```

Two implementations:

$$\texttt{foldl f z } \overrightarrow{[x_1,\ldots,x_n]} \;\cong\; f(x_n,\ldots f(x_3,f(x_2,f(x_1,z))))$$

$$\texttt{foldr f z } \overleftarrow{[x_1,\ldots,x_n]} \;\cong\; f(x_1,\ldots f(x_{n-2},f(x_{n-1},f(x_n,z))))$$

Examples:

```
foldl (op -) 0 [1,2,3,4] ==> 2 (4-(3-(2-(1-0))))
```

Higher-order function: fold

Combining elements in a list, given a binary operation and base value:

```
(* fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
```

Two implementations:

$$\texttt{foldl f z } \overrightarrow{[x_1,\ldots,x_n]} \cong \texttt{f}(x_n,\ldots\texttt{f}(x_3,\texttt{f}(x_2,\texttt{f}(x_1,z))))$$

$$\texttt{foldr f z } \overleftarrow{[x_1,\ldots,x_n]} \cong \texttt{f}(x_1,\ldots\texttt{f}(x_{n-2},\texttt{f}(x_{n-1},\texttt{f}(x_n,z))))$$

Examples:

```
foldl (op -) 0 [1,2,3,4] ==> 2
foldr (op -) 0 [1,2,3,4] ==>
```

Higher-order function: fold

Combining elements in a list, given a binary operation and base value:

```
(* fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
```

Two implementations:

$$\texttt{foldl f z } \overrightarrow{[x_1,\ldots,x_n]} \cong f(x_n,\ldots f(x_3,f(x_2,f(x_1,z))))$$

$$\texttt{foldr f z } \underleftarrow{[x_1,\ldots,x_n]} \cong f(x_1,\ldots f(x_{n-2},f(x_{n-1},f(x_n,z))))$$

Examples:

```
foldl (op -) 0 [1,2,3,4] ==> 2
foldr (op -) 0 [1,2,3,4] ==> ~2  (1-(2-(3-(4-0))))
```

Higher-order function: fold

Let's implement foldl and foldr:

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

Higher-order function: fold

Let's implement foldl and foldr:

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

Higher-order function: fold

Let’s implement foldl and foldr:

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

Higher-order function: fold

Let's implement foldl and foldr:

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

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

Higher-order function: fold

Let's implement foldl and foldr:

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

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

Higher-order function: fold

Let's implement foldl and foldr:

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

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

Homework:

```
foldl (op ::) [] [1,2,3,4] ==> ?
foldr (op ::) [] [1,2,3,4] ==> ?
```

That's all for today. See you on Thursday!