map

- **The most famous of all higher-order functions:**

```
let rec map f lis = if lis=[] then []
                    else (f (hd lis)) :: map f (tl lis);;
```

  - `map (fun x->x+1) [1;2;3]` = [2; 3; 4]
  - `let incrBy n lis = map (fun x -> x+n) lis`

  - `let incrBy n = map (fun x -> x+n)`

- **Type of `map`?** $(\alpha \to \beta) \to \alpha$ list $\to$ β list

map exercises

- **addpairs: (int * int) list → int list**

```
let addpairs = map (fun (x,y) -> x+y)
```

- **appendString: string → string list → string list concatenates the first argument to the end of every string in the second argument**

```
let appendString s = map (fun s' -> s'^s)
```

- **incrall: int list list → int list list increments every element of every list in its argument**

```
let incrall = map (map inc)
```

fold_right

- **Usually called `reduce`, but called `fold_right` in OCaml:**

```
let rec fold_right (f:'a->'b->'b) (lis:'a list) (z:'b) : 'b
      = if lis=[] then z else f (hd lis) (fold_right f (tl lis) z)
```

- `fold_right (fun s s' -> s @ s') ["a"; "b"; "c"] ""`

  = "abc"

- `fold_right (fun x y -> x+y) [3;4;5] 0`

  = 12

- `fold_right (fun x y -> x::y) [3;4;5] []`

  = [3;4;5]

- `let h f lis = fold_right (fun x y -> (f x)::y) lis []`

  = map f lis

Currying (cont.)

- **Can define functions** `curry` **and** `uncurry`**:**

```
let curry (f:'a * 'b -> 'c) : ('a -> 'b -> 'c) =
```

fun a → fun b → f(a,b)

```
let uncurry (f:'a -> 'b -> 'c) : ('a * 'b -> 'c) =
```

fun (a,b) → fun a b

```
usecurried (curry f_uncurried);;
useuncurried (uncurry f_curried);;
```

More h-o function examples

- `reverse (f: 'a -> 'b -> 'c) : ('b -> 'a -> 'c)`

  `reverse (fun x y -> x-y) 3 5 = 2`

  let reverse f = fun a → fun b → f b a

  or, let reverse f a b = f b a

- `twice (f: 'a -> 'a ) : ('a -> 'a)`

  `twice (fun x -> x+1) 3 = 5`

  let twice f = fun a → f (f a)

  or, let twice f a = f (f a)

Representing sets as functions

```
type intset = int -> bool

let emptyset : intset =
```

fun n → false

```
let member (n:int) (s:intset) : bool =
```

s n

```
let add (n:int) (s:intset) : intset =
```

fun n' → n' = n || s n'

Representing sets as functions (cont.)

```
let union (s1:intset) (s2:intset) : intset =
    fun n -> s1 n || s2 n

let intersection (s1:intset) (s2:intset) : intset =
    fun n -> s1 n && s2 n

let remove (n:int) (s:intset) : intset =
    fun n' -> s n' && n' <> n

let complement (s:intset) : intset =
    fun n -> not (s n)

let intsAbove (n:int) : intset =
    fun n' -> n' > n
```

Representing dictionaries as functions

- **Similarly, implementing an *environment* (a.k.a. *dictionary*), as in our mps, means defining a representation "`type environment =` *something*", and operations:**
  - `let emptyEnv : environment`
  - `fetch (id:id) (env:environment) : value`
  - `extend (id:id) (v:value) (env:environment) : environment`
- **These operations need to act in an "environment-like" way, e.g.**
  - `fetch "x" emptyEnv` **throws an exception**
  - `fetch "x" (extend "x" (IntConst 3) emptyEnv)` **returns** `IntConst 3`