Higher Order Programming: Folds

Concepts of Programming Languages
Lecture 10

CAS CS 320

Outline

- Look at one more common HOF in detail: **fold_left** (and **fold_right**)
- Look at HOFs on data types other than lists

Practice Problem

Implement the function

val smallest_prime_factor : int -> int

so that **smallest_prime_factor n** *is the smallest prime factor of* **n** *if* **n > 1**

Use this to define the predicate **p** *such that* **List.filter p l** *returns the elements of* **l** *which are the product of two distinct primes*

Recap

Recall: Definition of Map

```
let rec map f l =
  match l with
  | [] -> []
  | x :: xs -> f x :: map f xs
```

Recall: Definition of Map

```
let rec map f l =
  match l with
  | [] -> []
  | x :: xs -> f x :: map f xs
```

» *If the list is empty, nothing to do*

Recall: Definition of Map

```
let rec map f l =
  match l with
  | [] -> []
  | x :: xs -> f x :: map f xs
```

- *If the list is empty, nothing to do*
- *If the list is nonempty, we apply f to first element, and recurse*
- *Applies f to every element of l*

Recall: Filter

```
let rec filter p l =
  match l with
  | [] -> []
  | x :: xs ->
    (if p x then [x] else []) @ filter p xs
```

Recall: Filter

```
let rec filter p l =
  match l with
  | [] -> []
  | x :: xs ->
    (if p x then [x] else []) @ filter p xs
```

» *If the list is empty, nothing to do*

Recall: Filter

```
let rec filter p l =
  match l with
  | [] -> []
  | x :: xs ->
    (if p x then [x] else []) @ filter p xs
```

» *If the list is empty, nothing to do*

» *If the first element satisfies predicate, we keep it and recurse; otherwise, we drop it and recurse*

Recall: Filter

```
let rec filter p l =
  match l with
  | [] -> []
  | x :: xs ->
    (if p x then [x] else []) @ filter p xs
```

- » *If the list is empty, nothing to do*
- » *If the first element satisfies predicate, we keep it and recurse; otherwise, we drop it and recurse*

Recall: Filter

```
let rec filter p l =
  match l with
  | [] -> []
  | x :: xs ->
    (if p x then [x] else []) @ filter p xs
```

- *If the list is empty, nothing to do*
- *If the first element satisfies predicate, we keep it and recurse; otherwise, we drop it and recurse*

Recall: Filter

```
let rec filter p l =
  match l with
  | [] -> []
  | x :: xs ->
    (if p x then [x] else []) @ filter p xs
```

- *If the list is empty, nothing to do*
- *If the first element satisfies predicate, we keep it and recurse; otherwise, we drop it and recurse*
- *Keeps the elements of l that pass the filter p*

Folds

Overview

Overview

map	transform each element (keep every element)

Overview

`map`	transform each element (keep every element)
`filter`	keep some elements based on a predicate

Overview

map	transform each element (keep every element)
filter	keep some elements based on a predicate
fold	combine elements via an accumulation function

Observe the Pattern!

```
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

```
let rec concat ls =
  match ls with
  | [] -> []
  | x :: xs -> x @ concat xs
```

```
let rec rev l =
  match l with
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

```
let rec map f l =
  match l with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

Observe the Pattern!

```
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

```
let rec concat ls =
  match ls with
  | [] -> []
  | x :: xs -> x @ concat xs
```

```
let rec rev l =
  match l with
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

```
let rec map f l =
  match l with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

Observe the Pattern!

```
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

base

```
let rec concat ls =
  match ls with
  | [] -> []
  | x :: xs -> x @ concat xs
```

base

```
let rec rev l =
  match l with
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

base

```
let rec map f l =
  match l with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

base

Observe the Pattern!

```
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

base **rec. call**

```
let rec concat ls =
  match ls with
  | [] -> []
  | x :: xs -> x @ concat xs
```

base **rec. call**

```
let rec rev l =
  match l with
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

base **rec. call**

```
let rec map f l =
  match l with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

base **rec. call**

Observe the Pattern!

```
let rec sum l =
 match l with
 | [] -> 0
 | x :: xs -> x + sum xs
```

base rec. call combine

```
let rec concat ls =
 match ls with
 | [] -> []
 | x :: xs -> x @ concat xs
```

base rec. call combine

```
let rec rev l =
 match l with
 | [] -> []
 | x :: xs -> rev xs @ [x]
```

base rec. call combine

```
let rec map f l =
 match l with
 | [] -> []
 | x :: xs -> (f x) :: map f xs
```

base rec. call combine

Fold as Specialized Pattern Matching

```
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

Fold as Specialized Pattern Matching

```
let rec sum l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum xs
```

Fold as Specialized Pattern Matching

```
let rec sum l =
  let base = 0 in
  match l with
  | [] -> base
  | x :: xs -> x + sum xs
```

Fold as Specialized Pattern Matching

```
let rec sum l =
  let base = 0 in
  match l with
  | [] -> base
  | x :: xs -> x + sum xs
```

Fold as Specialized Pattern Matching

```
let rec sum l =
  let base = 0 in
  let op = (+) in
  match l with
  | [] -> base
  | x :: xs -> op x (sum xs)
```

Fold as Specialized Pattern Matching

```
let rec sum l =
  let base = 0 in
  let op = (+) in
  match l with
  | [] -> base
  | x :: xs -> op x (sum xs)
```

Fold as Specialized Pattern Matching

```
let sum l =
  let base = 0 in
  let op = (+) in
  let rec go op l base =
    match l with
    | [] -> base
    | x :: xs -> op x (go xs)
  in go op l base
```

Fold as Specialized Pattern Matching

```
let sum l =
  let base = 0 in
  let op = (+) in
  let rec go op l base =
    match l with
    | [] -> base
    | x :: xs -> op x (go xs)
  in go op l base
```

fold right

Fold as Specialized Pattern Matching

```
let sum l =
  let base = 0 in
  let op = ( + ) in
  List.fold_right op l base
```

Fold as Specialized Pattern Matching

```
let sum l = List.fold_right ( + ) l 0
```

We get a one–liner for **sum** (and a whole lot of other functions)

Folds are very nice for "iterating" over a list

The Picture

```
1 :: (2 :: (3 :: (4 :: (5 :: (6 :: (7 :: [])))))))
```

$\downarrow$ `sum = fold_right ( + ) l 0`

```
1 +  (2 +  (3 +  (4 +  (5 +  (6 +  (7 +  0 )))))))
```

We can think of **fold_right** as "replacing" :: with + and [] with 0

The Picture

```
1 :: (2 :: (3 :: (4 :: (5 :: (6 :: (7 :: [])))))))

        prod = fold_right ( * ) l 1

1 *  (2 *  (3 *  (4 *  (5 *  (6 *  (7 *  1 )))))))
```

We can think of **fold_right** as "replacing" :: with * and [] with 1

The Picture

```
[1] :: ([2] :: ([3] :: ([4] :: ([5] :: ([6] :: ([7] :: [])))))))
```

$\downarrow$ `concat = fold_right ( @ ) l []`

```
[1] @  ([2] @  ([3] @  ([4] @  ([5] @  ([6] @  ([7] @  [])))))))
```

We can think of **fold_right** as "replacing" `::` with `@` and `[]` with `[]`

The Picture

```
1 :: (2 :: (3 :: (4 :: (5 :: (6 :: (7 :: [])))))))
```

$$\Big\downarrow \texttt{fold\_right op l base}$$

```
op 1 (op 2 (op 3 (op 4 (op 5 (op 6 (op 7 base))))))
```

We can think of **fold_right** as "replacing" :: with op and [] with base

Definition of Fold Right

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

```
let fold_right op l base =
  let rec go l =
    match l with
    | [] -> base
    | x :: xs -> op x (go xs)
  in go l
```

Definition of Fold Right

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b note the order of args.

```
let fold_right op l base =
  let rec go l =
    match l with
    | [] -> base
    | x :: xs -> op x (go xs)
  in go l
```

Definition of Fold Right

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b   note the order of args.

```
let fold_right op l base =
  let rec go l =
    match l with
    | [] -> base
    | x :: xs -> op x (go xs)
  in go l
```

» On empty, return the **base** element

Definition of Fold Right

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b note the order of args.

```
let fold_right op l base =
  let rec go l =
    match l with
    | [] -> base
    | x :: xs -> op x (go xs)
  in go l
```

- On empty, return the **base** element
- On nonempty, recurse on the tail and apply **op** to the head and the result

Definition of Fold Right

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b note the order of args.

```
let fold_right op l base =
  let rec go l =
    match l with
    | [] -> base
    | x :: xs -> op x (go xs)
  in go l
```

Is this tail recursive?

» On empty, return the **base** element

» On nonempty, recurse on the tail and apply **op** to the head and the result

Practice Problem

Using **List.fold_right**, implement

» **append (@)**

» **reverse**

» **map**

» **filter**

demo

(examples of fold_right)

Tail-Recursive Fold Attempt

```
let fold_right_tr op l base =
  let rec go l acc =
    match l with
    | [] -> acc
    | x :: xs -> go xs (op acc x)
  in go l base
```

Can you see what's wrong with this definition?

The Problem

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0            ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0              ===
1 + fold_right (+) [2;3] 0            ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0          ===
1 + fold_right (+) [2;3] 0        ===
1 + (2 + fold_right (+) [3] 0)    ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0             ===
1 + fold_right (+) [2;3] 0           ===
1 + (2 + fold_right (+) [3] 0)       ===
1 + (2 + (3 + fold_right (+) [] 0)) ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0              ===
1 + fold_right (+) [2;3] 0            ===
1 + (2 + fold_right (+) [3] 0)        ===
1 + (2 + (3 + fold_right (+) [] 0)) ===
1 + (2 + (3 + 0))                     ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0             ===
1 + fold_right (+) [2;3] 0           ===
1 + (2 + fold_right (+) [3] 0)       ===
1 + (2 + (3 + fold_right (+) [] 0)) ===
1 + (2 + (3 + 0))                    ===
1 + (2 + 3)                          ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0              ===
1 + fold_right (+) [2;3] 0            ===
1 + (2 + fold_right (+) [3] 0)        ===
1 + (2 + (3 + fold_right (+) [] 0))   ===
1 + (2 + (3 + 0))                     ===
1 + (2 + 3)                           ===
1 + 5                                 ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0              ===
1 + fold_right (+) [2;3] 0            ===
1 + (2 + fold_right (+) [3] 0)        ===
1 + (2 + (3 + fold_right (+) [] 0)) ===
1 + (2 + (3 + 0))                     ===
1 + (2 + 3)                           ===
1 + 5                                 ===
6
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0                ===
1 + fold_right (+) [2;3] 0              ===
1 + (2 + fold_right (+) [3] 0)          ===
1 + (2 + (3 + fold_right (+) [] 0)) ===
1 + (2 + (3 + 0))                       ===
1 + (2 + 3)                             ===
1 + 5                                   ===
6
```

```
fold_right_tr (+) [1;2;3] 0   ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0                  ===
1 + fold_right (+) [2;3] 0                ===
1 + (2 + fold_right (+) [3] 0)            ===
1 + (2 + (3 + fold_right (+) [] 0)) ===
1 + (2 + (3 + 0))                         ===
1 + (2 + 3)                               ===
1 + 5                                     ===
6
```

```
fold_right_tr (+) [1;2;3] 0   ===
go [1;2;3] 0                  ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0                     ===
1 + fold_right (+) [2;3] 0                   ===
1 + (2 + fold_right (+) [3] 0)               ===
1 + (2 + (3 + fold_right (+) [] 0))          ===
1 + (2 + (3 + 0))                            ===
1 + (2 + 3)                                  ===
1 + 5                                        ===
6
```

```
fold_right_tr (+) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 + 1)              ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0                 ===
1 + fold_right (+) [2;3] 0               ===
1 + (2 + fold_right (+) [3] 0)           ===
1 + (2 + (3 + fold_right (+) [] 0))      ===
1 + (2 + (3 + 0))                        ===
1 + (2 + 3)                              ===
1 + 5                                    ===
6
```

```
fold_right_tr (+) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 + 1)              ===
go [3] ((0 + 1) + 2)          ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0                 ===
1 + fold_right (+) [2;3] 0               ===
1 + (2 + fold_right (+) [3] 0)           ===
1 + (2 + (3 + fold_right (+) [] 0))      ===
1 + (2 + (3 + 0))                        ===
1 + (2 + 3)                              ===
1 + 5                                    ===
6
```

```
fold_right_tr (+) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 + 1)              ===
go [3] ((0 + 1) + 2)          ===
go [] (((0 + 1) + 2) + 3)     ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0                 ===
1 + fold_right (+) [2;3] 0               ===
1 + (2 + fold_right (+) [3] 0)           ===
1 + (2 + (3 + fold_right (+) [] 0))      ===
1 + (2 + (3 + 0))                        ===
1 + (2 + 3)                              ===
1 + 5                                    ===
6
```

```
fold_right_tr (+) [1;2;3] 0    ===
go [1;2;3] 0                   ===
go [2;3] (0 + 1)               ===
go [3] ((0 + 1) + 2)           ===
go [] (((0 + 1) + 2) + 3)      ===
((0 + 1) + 2) + 3              ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0                  ===
1 + fold_right (+) [2;3] 0                ===
1 + (2 + fold_right (+) [3] 0)            ===
1 + (2 + (3 + fold_right (+) [] 0))       ===
1 + (2 + (3 + 0))                         ===
1 + (2 + 3)                               ===
1 + 5                                     ===
6
```

```
fold_right_tr (+) [1;2;3] 0    ===
go [1;2;3] 0                   ===
go [2;3] (0 + 1)               ===
go [3] ((0 + 1) + 2)           ===
go [] (((0 + 1) + 2) + 3)      ===
((0 + 1) + 2) + 3              ===
(1 + 2) + 3                    ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0                ===
1 + fold_right (+) [2;3] 0              ===
1 + (2 + fold_right (+) [3] 0)          ===
1 + (2 + (3 + fold_right (+) [] 0)) ===
1 + (2 + (3 + 0))                       ===
1 + (2 + 3)                             ===
1 + 5                                   ===
6
```

```
fold_right_tr (+) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 + 1)              ===
go [3] ((0 + 1) + 2)          ===
go [] (((0 + 1) + 2) + 3)     ===
((0 + 1) + 2) + 3             ===
(1 + 2) + 3                   ===
3 + 3                         ===
```

Note: this is not the order of operations, it is just for illustration

The Problem

```
fold_right (+) [1;2;3] 0                 ===
1 + fold_right (+) [2;3] 0               ===
1 + (2 + fold_right (+) [3] 0)           ===
1 + (2 + (3 + fold_right (+) [] 0))      ===
1 + (2 + (3 + 0))                        ===
1 + (2 + 3)                              ===
1 + 5                                    ===
6
```

```
fold_right_tr (+) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 + 1)              ===
go [3] ((0 + 1) + 2)          ===
go [] (((0 + 1) + 2) + 3)     ===
((0 + 1) + 2) + 3             ===
(1 + 2) + 3                   ===
3 + 3                         ===
6
```

Note: this is not the order of operations, it is just for illustration

The Problem

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0          ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0          ===
1 - fold_right (-) [2;3] 0        ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0            ===
1 - fold_right (-) [2;3] 0          ===
1 - (2 - fold_right (-) [3] 0)      ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0             ===
1 - fold_right (-) [2;3] 0           ===
1 - (2 - fold_right (-) [3] 0)       ===
1 - (2 - (3 - fold_right (-) [] 0)) ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0               ===
1 - fold_right (-) [2;3] 0             ===
1 - (2 - fold_right (-) [3] 0)         ===
1 - (2 - (3 - fold_right (-) [] 0))    ===
1 - (2 - (3 - 0))                      ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0          ===
1 - fold_right (-) [2;3] 0        ===
1 - (2 - fold_right (-) [3] 0)    ===
1 - (2 - (3 - fold_right (-) [] 0)) ===
1 - (2 - (3 - 0))                 ===
1 - (2 - 3)                       ===
```

∗Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0                 ===
1 - fold_right (-) [2;3] 0               ===
1 - (2 - fold_right (-) [3] 0)           ===
1 - (2 - (3 - fold_right (-) [] 0))      ===
1 - (2 - (3 - 0))                        ===
1 - (2 - 3)                              ===
1 - (-1)                                 ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0               ===
1 - fold_right (-) [2;3] 0             ===
1 - (2 - fold_right (-) [3] 0)         ===
1 - (2 - (3 - fold_right (-) [] 0))    ===
1 - (2 - (3 - 0))                      ===
1 - (2 - 3)                            ===
1 - (-1)                               ===
2
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0              ===
1 - fold_right (-) [2;3] 0            ===
1 - (2 - fold_right (-) [3] 0)        ===
1 - (2 - (3 - fold_right (-) [] 0)) ===
1 - (2 - (3 - 0))                     ===
1 - (2 - 3)                           ===
1 - (-1)                              ===
2
```

```
fold_right_tr (-) [1;2;3] 0   ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0                 ===
1 - fold_right (-) [2;3] 0               ===
1 - (2 - fold_right (-) [3] 0)           ===
1 - (2 - (3 - fold_right (-) [] 0))      ===
1 - (2 - (3 - 0))                        ===
1 - (2 - 3)                              ===
1 - (-1)                                 ===
2
```

```
fold_right_tr (-) [1;2;3] 0   ===
go [1;2;3] 0                  ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0               ===
1 - fold_right (-) [2;3] 0             ===
1 - (2 - fold_right (-) [3] 0)         ===
1 - (2 - (3 - fold_right (-) [] 0)) ===
1 - (2 - (3 - 0))                      ===
1 - (2 - 3)                            ===
1 - (-1)                               ===
2
```

```
fold_right_tr (-) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 - 1)              ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0              ===
1 - fold_right (-) [2;3] 0            ===
1 - (2 - fold_right (-) [3] 0)        ===
1 - (2 - (3 - fold_right (-) [] 0)) ===
1 - (2 - (3 - 0))                     ===
1 - (2 - 3)                           ===
1 - (-1)                              ===
2
```

```
fold_right_tr (-) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 - 1)              ===
go [3] ((0 - 1) - 2)          ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0                 ===
1 - fold_right (-) [2;3] 0               ===
1 - (2 - fold_right (-) [3] 0)           ===
1 - (2 - (3 - fold_right (-) [] 0))      ===
1 - (2 - (3 - 0))                        ===
1 - (2 - 3)                              ===
1 - (-1)                                 ===
2
```

```
fold_right_tr (-) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 - 1)              ===
go [3] ((0 - 1) - 2)          ===
go [] (((0 - 1) - 2) - 3)     ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0                ===
1 - fold_right (-) [2;3] 0              ===
1 - (2 - fold_right (-) [3] 0)          ===
1 - (2 - (3 - fold_right (-) [] 0))     ===
1 - (2 - (3 - 0))                       ===
1 - (2 - 3)                             ===
1 - (-1)                                ===
2
```

```
fold_right_tr (-) [1;2;3] 0    ===
go [1;2;3] 0                   ===
go [2;3] (0 - 1)               ===
go [3] ((0 - 1) - 2)           ===
go [] (((0 - 1) - 2) - 3)      ===
((0 - 1) - 2) - 3              ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0             ===
1 - fold_right (-) [2;3] 0           ===
1 - (2 - fold_right (-) [3] 0)       ===
1 - (2 - (3 - fold_right (-) [] 0))  ===
1 - (2 - (3 - 0))                    ===
1 - (2 - 3)                          ===
1 - (-1)                             ===
2
```

```
fold_right_tr (-) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 - 1)              ===
go [3] ((0 - 1) - 2)          ===
go [] (((0 - 1) - 2) - 3)     ===
((0 - 1) - 2) - 3             ===
((-1) - 2) - 3                ===
```

∗Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0               ===
1 - fold_right (-) [2;3] 0             ===
1 - (2 - fold_right (-) [3] 0)         ===
1 - (2 - (3 - fold_right (-) [] 0))    ===
1 - (2 - (3 - 0))                      ===
1 - (2 - 3)                            ===
1 - (-1)                               ===
2
```

```
fold_right_tr (-) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 - 1)              ===
go [3] ((0 - 1) - 2)          ===
go [] (((0 - 1) - 2) - 3)     ===
((0 - 1) - 2) - 3             ===
((-1) - 2) - 3                ===
(-3) - 3                      ===
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0                ===
1 - fold_right (-) [2;3] 0              ===
1 - (2 - fold_right (-) [3] 0)          ===
1 - (2 - (3 - fold_right (-) [] 0))     ===
1 - (2 - (3 - 0))                       ===
1 - (2 - 3)                             ===
1 - (-1)                                ===
2
```

```
fold_right_tr (-) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 - 1)              ===
go [3] ((0 - 1) - 2)          ===
go [] (((0 - 1) - 2) - 3)     ===
((0 - 1) - 2) - 3             ===
((-1) - 2) - 3                ===
(-3) - 3                      ===
-6
```

*Note: this is *not* the order of operations, it is just for illustration

The Problem

```
fold_right (-) [1;2;3] 0               ===
1 - fold_right (-) [2;3] 0             ===
1 - (2 - fold_right (-) [3] 0)         ===
1 - (2 - (3 - fold_right (-) [] 0))    ===
1 - (2 - (3 - 0))                      ===
1 - (2 - 3)                            ===
1 - (-1)                               ===
2
```

$$1-(2-(3-0))$$

```
fold_right_tr (-) [1;2;3] 0   ===
go [1;2;3] 0                  ===
go [2;3] (0 - 1)              ===
go [3] ((0 - 1) - 2)          ===
go [] (((0 - 1) - 2) - 3)     ===
((0 - 1) - 2) - 3             ===
((-1) - 2) - 3                ===
(-3) - 3                      ===
-6
```

$$((0-1)-2)-3$$

Changing parentheses is fine for (+) but not for (-)

*Note: this is *not* the order of operations, it is just for illustration

Associativity

<u>Definition:</u> A binary operation $\square : A \times A \to A$ is **associative** if it satisfies

$$a \square (b \square c) = (a \square b) \square c$$

for any $a, b, c \in A$

<u>Example:</u> Addition and multiplication are associative, whereas subtraction and division are not

Definition of Fold Left

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

```
let fold_left op base l =
  let rec go l acc =
    match l with
    | [] -> acc
    | x :: xs -> go xs (op acc x)
  in go l base
```

Definition of Fold Left

('a -> 'b -> 'a) -> 'a -> 'b list -> 'a   note the order of args.

```
let fold_left op base l =
  let rec go l acc =
    match l with
    | [] -> acc
    | x :: xs -> go xs (op acc x)
  in go l base
```

Folding left is just our incorrect tail recursive right folding (with a change in the order of arguments)

Definition of Fold Left

('a -> 'b -> 'a) -> 'a -> 'b list -> 'a note the order of args.

```
let fold_left op base l =
  let rec go l acc =
    match l with
    | [] -> acc
    | x :: xs -> go xs (op acc x)
  in go l base
```

Folding left is just our incorrect tail recursive right folding (with a change in the order of arguments)

fold_left is a left-associative fold
fold_right is a right-associative fold

The Picture

```
1 :: (2 :: (3 :: (4 :: [])))
```

fold_right op l base

```
op 1 (op 2 (op 3 (op 4 base)))
```

fold_left op base l

```
op (op (op (op base 1) 2) 3) 4
```

The Picture

`1 :: (2 :: (3 :: (4 :: [])))`

`fold_left op base l`

`fold_right op l base`

`op 1 (op 2 (op 3 (op 4 base)))`

`op (op (op (op base 1) 2) 3) 4`

demo

(examples of fold_left)

Examples of fold_left

```
('a -> bool) -> 'a list -> 'a list
let filter p l = fold_left (fun res h -> if p h then res @ [h] else res) [] l
```

```
'a list -> 'a list -> 'a list
let append l1 l2 = fold_left (fun res h -> res @ [h]) l1 l2
```

```
'a list -> 'a list
let reverse l = fold_left (fun res h -> h::res) [] l
```

```
int list -> int
let max l = fold_left (fun m h -> if h > m then h else m) 0 l
```

Tail-Recursive Fold Right

```
let fold_right_tr op l base =
  List.fold_left
    (fun x y -> op y x)
    base
    (List.rev l)
```

We can write fold_right in terms of fold left by reversing the list and "reversing" the operation

Challenge: *Write a tail-recursive fold right without reversing the list*

The Picture

Let **x** $-_r$ **y** := **y** $-$ **x**, subtraction with the arguments flipped

The Picture

Let **x** `-r` **y** := **y** `-` **x**, subtraction with the arguments flipped

```
1 -r (2 -r (3 -r (4 -r 0)))
```

The Picture

Let **x** `-r` **y** := **y** `-` **x**, subtraction with the arguments flipped

```
  1 -r (2 -r (3 -r (4 -r 0)))
= 1 -r (2 -r (3 -r (0 - 4)))
```

The Picture

Let **x** -r **y** := **y** - **x**, subtraction with the arguments flipped

```
  1 -r (2 -r (3 -r (4 -r 0)))
= 1 -r (2 -r (3 -r (0 - 4)))
= 1 -r (2 -r ((0 - 4) - 3))
```

The Picture

Let **x** -r **y** := **y** - **x**, subtraction with the arguments flipped

```
  1 -r (2 -r (3 -r (4 -r 0)))
= 1 -r (2 -r (3 -r (0 - 4)))
= 1 -r (2 -r ((0 - 4) - 3))
= 1 -r (((0 - 4) - 3) - 2)
```

The Picture

Let **x** -r **y** := **y** - **x**, subtraction with the arguments flipped

```
  1 -r (2 -r (3 -r (4 -r 0)))
= 1 -r (2 -r (3 -r (0 - 4)))
= 1 -r (2 -r ((0 - 4) - 3))
= 1 -r (((0 - 4) - 3) - 2)
= (((0 - 4) - 3) - 2) - 1
```

Short Circuiting

```
let rec all bs =
  match bs with
    | [] -> true
    | false :: _ -> false
    | true :: t -> all t

let all = List.fold_left (&&) true
```

Short Circuiting

```
let rec all bs =
  match bs with
    | [] -> true
    | false :: _ -> false
    | true :: t -> all t

let all = List.fold_left (&&) true
```

Which is better?

Short Circuiting

```
let rec all bs =
  match bs with
    | [] -> true
    | false :: _ -> false
    | true :: t -> all t

let all = List.fold_left (&&) true
```

Which is better?

fold_left has to traverse the entire list, it can't short-circuit

Short Circuiting

```
let rec all bs =
  match bs with
    | [] -> true
    | false :: _ -> false
    | true :: t -> all t

let all = List.fold_left (&&) true
```

Which is better?

fold_left has to traverse the entire list, it can't short-circuit

But the fold code is shorter and arguably clearer...

General Rules for Folds

General Rules for Folds

- For **associative** operations, use **fold_left.** Why? Because it is tail recursive

General Rules for Folds

» For **associative** operations, use **fold_left.** Why? Because it is tail recursive

» Otherwise think! If the recursive call is to be made on the tail, use **fold_right**

General Rules for Folds

- For **associative** operations, use **fold_left.** Why? Because it is tail recursive
- Otherwise think! If the recursive call is to be made on the tail, use **fold_right**
- If you need an accumulator from left to right, use **fold_left**

General Rules for Folds

- For **associative** operations, use **fold_left.** Why? Because it is tail recursive
- Otherwise think! If the recursive call is to be made on the tail, use **fold_right**
- If you need an accumulator from left to right, use **fold_left**
- The types are difficult to remember, let the compiler remind you

Practice Problem

```
let rec insert le v l =
  match l with
  | [] -> [v]
  | x :: xs ->
    if le v x
    then v :: l
    else x :: insert le v l
```

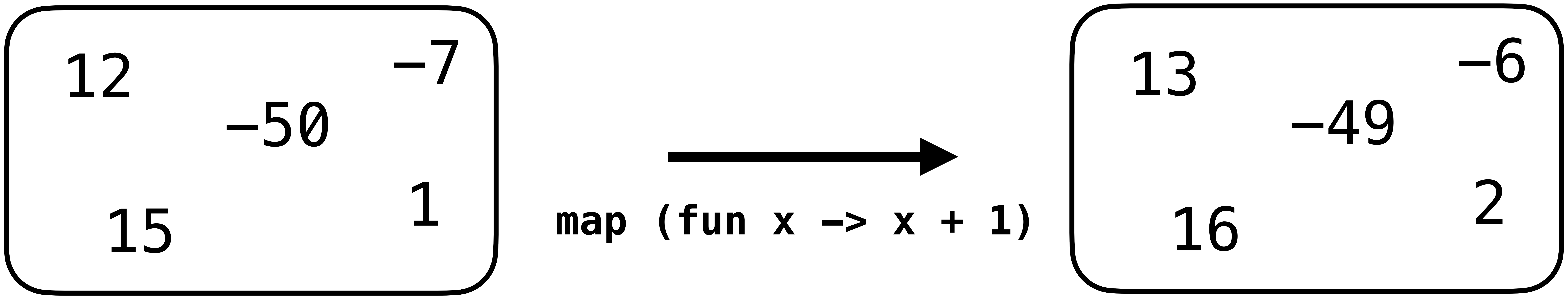
In terms of **fold_left** implement the function

val sort : ('a -> 'a -> bool) -> 'a list -> 'a list

so that **sort le l** is the list **l** in sorted order according to **le**

Beyond Lists

Mappable Data



A lot of data types hold uniform kinds of data which can then be mapped over

Formally, these are called **Functors**

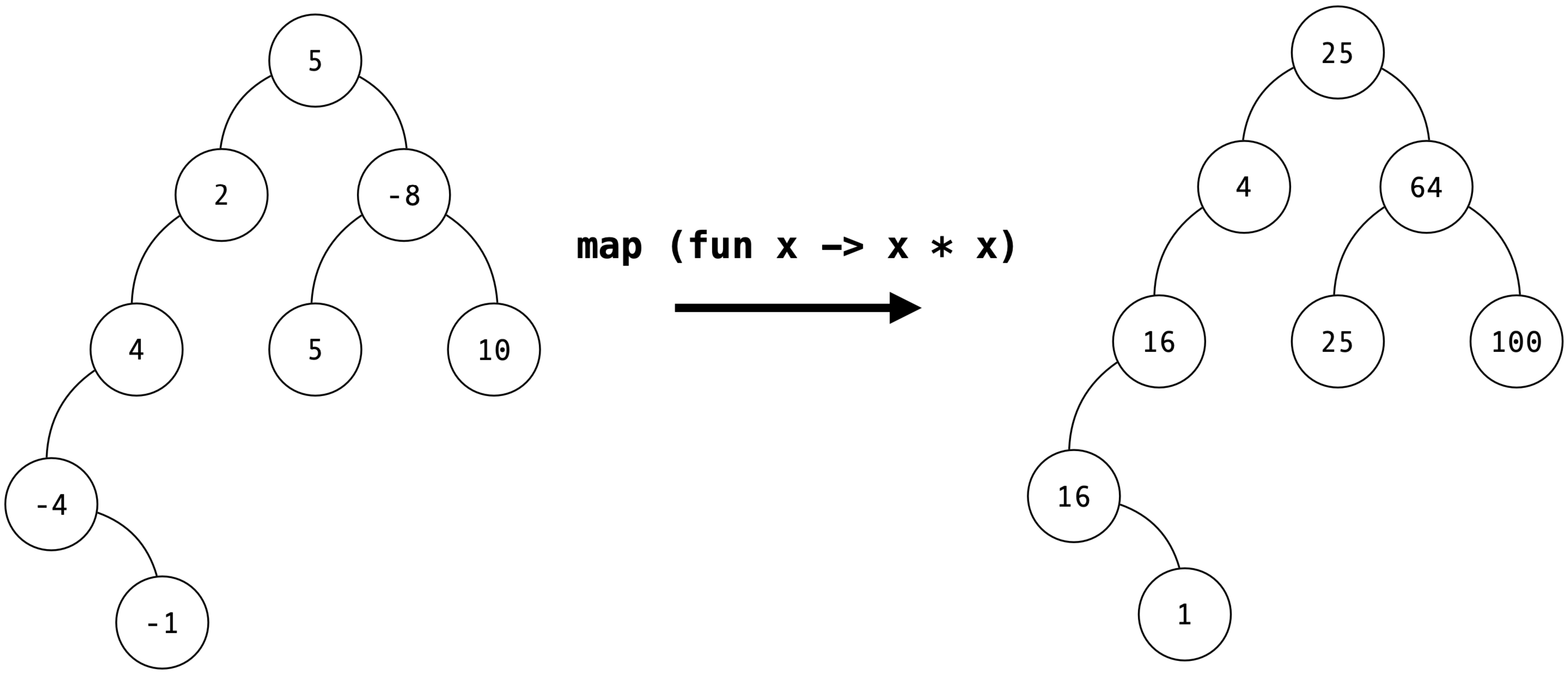
Trees

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree

let map f t =
  let rec go t =
    match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, go l, go r)
  in go t
```

Mapping over a tree maintains the structure but recursively updates values with **f**

The Picture



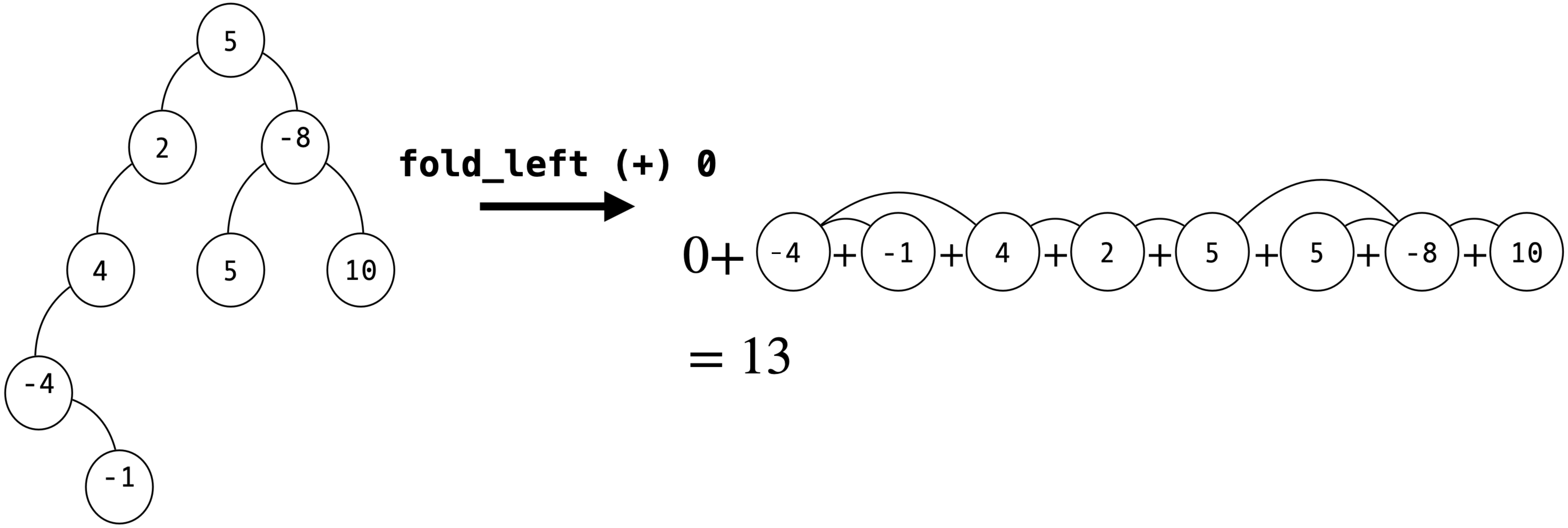
Options

```
let map f oa =
 let rec go oa =
 match oa with
   | None -> None
   | Some x -> Some (f x)
 in go oa
```

*On **None**, leave the **None***

*On **Some x**, apply **f** to **x***

Trees (The Picture)



In-Order Fold for Trees

```
let fold_in_order op base t =
  let rec go acc t =
    match t with
    | Leaf -> acc
    | Node (x, l, r) -> go (op (go acc l) x) r
  in go base t
```

In-Order Fold for Trees

```
let fold_in_order op base t =
  let rec go acc t =
    match t with
    | Leaf -> acc
    | Node (x, l, r) -> go (op (go acc l) x) r
  in go base t
```

This is an **in-order** fold for trees

In-Order Fold for Trees

```
let fold_in_order op base t =
  let rec go acc t =
    match t with
    | Leaf -> acc
    | Node (x, l, r) -> go (op (go acc l) x) r
  in go base t
```

This is an **in-order** fold for trees

It is equivalent to "flattening" the tree into a list, and then folding that list

In-Order Fold for Trees

```
let fold_in_order op base t =
  let rec go acc t =
    match t with
    | Leaf -> acc
    | Node (x, l, r) -> go (op (go acc l) x) r
  in go base t
```

This is an **in-order** fold for trees

It is equivalent to "flattening" the tree into a list, and then folding that list

(This is different from what is given in the textbook)

In-Order Fold for Trees

```
let fold_in_order op base t =
 let rec go acc t =
  match t with
   | Leaf -> acc
   | Node (x, l, r) -> go (op (go acc l) x) r
 in go base t
```

not tail recursive

This is an **in-order** fold for trees

It is equivalent to "flattening" the tree into a list, and then folding that list

(This is different from what is given in the textbook)

demo

(map/fold over trees)

Summary

Folds are used to **combine** data with an accumulation function

The order that we combine things matters if the accumulation function is not **associative**

We can map and fold (and even filter) more than just lists