Higher-Order Programming II: Folds and Examples

Principles of Programming Languages

Lecture 9

CAS CS 320

Introduction

Administrivia

Assignment 4 is due on Friday by 11:59PM.

There is no assignment this week.

The midterm is next week 2/27 during class. **There will be two locations** (more details on Piazza this week).

Objectives

Discuss our last important higher-order function: folds.

Look at higher-order functions on data types beyond lists.

(this material can appear on the midterm)

Keywords

fold right

fold left

tail-recursion

associativity

mapping trees

folding trees

Practice Problem

*Implement a function **smallest_prime_factor** which, given (**n : int**), returns the smallest prime factor of **n** if **n > 1** and **1** otherwise.*

*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: Higher-Order Programming

Recall: Higher-Order Programming

In OCaml, functions are **first-class values:**

Recall: Higher-Order Programming

In OCaml, functions are **first-class values:**

1. They can be given names with let-definitions.

Recall: Higher-Order Programming

In OCaml, functions are **first-class values:**

1. They can be given names with let-definitions.
2. They can be returned by another function.

Recall: Higher-Order Programming

In OCaml, functions are **first-class values:**

1. They can be given names with let-definitions.
2. They can be returned by another function.
3. They can be passed as arguments to another function.

Recall: Higher-Order Programming

In OCaml, functions are **first-class values:**

1. They can be given names with let-definitions.
2. They can be returned by another function.
3. They can be passed as arguments to another function.

Note. Types are ***not*** first-class values.

Recall: Functions as Parameters

```
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
# apply add_five 10;;
- : int = 15
```

Recall: Functions as Parameters

```
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
# apply add_five 10;;
- : int = 15
```

This allows us to create new functions which are parametrized by old ones.

Recall: Functions as Parameters

```
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
# apply add_five 10;;
- : int = 15
```

note the type

This allows us to create new functions which are parametrized by old ones.

Recall: A Simple Example

```
let rec fact n =
  match n with
  | 0 -> 1
  | n -> n * fact (n - 1)

let rec sum n =
  match n with
  | 0 -> 0
  | n -> n + sum (n - 1)
```

Recall: A Simple Example

```
let rec fact n =
  match n with
  | 0 -> 1
  | n -> n * fact (n - 1)

let rec sum n =
  match n with
  | 0 -> 0
  | n -> n + sum (n - 1)
```

Can we abstract the core functionality?

Recall: A Simple Example

```
let rec fact n =
  match n with
  | 0 -> 1
  | n -> n * fact (n - 1)

let rec sum n =
  match n with
  | 0 -> 0
  | n -> n + sum (n - 1)
```

Can we abstract the core functionality?

Recall: A Simple Example

```
let rec upto f n start =
  let rec go n =
    match n with
    | 0 -> start
    | n -> f n (go (n - 1))
  in go n
```

Recall: A Simple Example

```
let rec upto f n start =
  let rec go n =
    match n with
    | 0 -> start
    | n -> f n (go (n - 1))
  in go n
```

In order to generalize this function, we need to be able to take the operation as a parameter.

Recall: A Simple Example

```
let rec upto f n start =
  let rec go n =
    match n with
    | 0 -> start
    | n -> f n (go (n - 1))
  in go n
```

In order to generalize this function, we need to be able to take the operation as a parameter.

Recall: A Simple Example

```
let rec upto f n start =
  let rec go n =
    match n with
    | 0 -> start
    | n -> f n (go (n - 1))
  in go n
```

In order to generalize this function, we need to be able to take the operation as a parameter.

Now we have a single function which we can reuse elsewhere.

Folds

An Overview

An Overview

```
map          transform each element of a list
             keeping every element
```

An Overview

`map`	`transform each element of a list keeping every element`
`filter`	`keep some elements, throw some away`

An Overview

`map`	`transform each element of a list keeping every element`
`filter`	`keep some elements, throw some away`
`fold`	`combine elements via an accumulation function`

A Couple Functions

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

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

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

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

A Couple Functions

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

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

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

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

A Couple Functions

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

base

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

base

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

base

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

base

A Couple Functions

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

base recursive call on tail

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

base recursive call on tail

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

base recursive call on tail

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

base recursive call on tail

A Couple Functions

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

base recursive call on tail

computation on hd and rec on tl

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

base recursive call on tail

computation on hd and rec on tl

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

base recursive call on tail

computation on hd and rec on tl

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

base recursive call on tail

computation on hd and rec on tl

Folding as Specialized Pattern Matching

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

Folding as Specialized Pattern Matching

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

Folding as Specialized Pattern Matching

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

Folding as Specialized Pattern Matching

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

Folding as Specialized Pattern Matching

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

fold_right

Folding as Specialized Pattern Matching

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

Folding as Specialized Pattern Matching

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

The Picture

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

$$\Big\downarrow \texttt{sum}$$

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

The Picture

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

sum

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

We can think of **fold_right** as "replacing" every ':' with '**op**'

The Picture

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

$$\Big\downarrow \texttt{sum}$$

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

We can think of **fold_right** as "replacing" every '::' with '**op**'

'**+**' in the case of **sum**.

The Picture

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

concat

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

We can think of **fold_right** as replacing every '::' with '**op**'

'**@**' in the case of **concat**.

The Picture

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

```
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 every '::' with '**op**'.

Fold Right

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

Fold Right

```
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.*

Fold Right

```
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.*

Fold Right

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

Note the order of arguments

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

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

Understanding Check

Write ***filter*** *using* ***List.fold_right.***

Write ***append (@)*** *using* ***List.fold_right.***

Tail-Recursive Fold Right (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

let _ = assert (fold_right_tr (+) [1;2;3;4;5] 0 = 15)
let _ = assert (fold_right_tr (@) [[1];[2];[3];[4]] [] = [1;2;3;4])
```

Tail-Recursive Fold Right (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

let _ = assert (fold_right_tr (+) [1;2;3;4;5] 0 = 15)
let _ = assert (fold_right_tr (@) [[1];[2];[3];[4]] [] = [1;2;3;4])
```

Question. *What's wrong with this?*

The Problem

The Problem

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

The Problem

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

The Problem

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

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)) ==>
```

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))                       ==>
```

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)                             ==>
```

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                                   ==>
```

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
```

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   ==>
```

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                  ==>
```

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)              ==>
```

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 [2;3] 1                    ==>
```

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

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

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

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

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

The Problem

The Problem

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

The Problem

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

The Problem

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

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    ==>
```

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))                     ==>
```

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)                             ==>
```

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)                               ==>
```

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
```

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   ==>
```

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                  ==>
```

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)              ==>
```

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 [2;3] (-1)                 ==>
```

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

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

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

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

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

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

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

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

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

Changing parentheses works for (+) but not for (-)

Associativity

Associativity

Definition. 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 $a, b, c \in A$.

Associativity

Definition. 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 $a, b, c \in A$.

Question. *What is another example of an associative operation? What about non-associative?*

Fold Left

```
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
```

Fold Left

```
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
```

Note the order of arguments

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

Fold Left

```
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
```

Note the order of arguments

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

Folding left is tail recursive by definition.

Fold Left

```
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
```

Note the order of arguments

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

Folding left is tail recursive by definition.

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_right op l base`

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

`fold_left op base l`

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

The Picture

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

`fold_right op l base`

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

`fold_left op base l`

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

Aside: Actually Tail-Recursive Fold Right

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

Aside: Actually 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.

Aside: Actually 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.

We have to reverse the list and "reverse" the operation.

Aside: Actually 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.

We have to reverse the list and "reverse" the operation.

Challenge. *Write a tail-recursive fold_right* ***without*** *using List.rev.*

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
```

Aside: Short Circuiting

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

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

Aside: Short Circuiting

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

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

Question. *Which is better?*

Aside: Short Circuiting

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

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

Question. *Which is better?*

fold_left *must* process all elements of a list, it cannot short circuit.

Aside: Short Circuiting

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

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

Question. *Which is better?*

fold_left *must* process all elements of a list, it cannot short circuit.

(Is this actually better though?)

General Rules for Folding

General Rules for Folding

For associative operations, use **fold_left**.

General Rules for Folding

For associative operations, use **fold_left**.

The types are difficult to remember, let your editor (or the compiler) remind you.

General Rules for Folding

For associative operations, use **fold_left**.

The types are difficult to remember, let your editor (or the compiler) remind you.

Don't use folds for everything, don't use pattern matching for everything. Think about the use case.

Understanding Check

Implement the function which finds the maximum element in a list, given an arbitrary '≤' function of type ***'a -> 'a -> bool.***

Write it with pattern matching and folds.

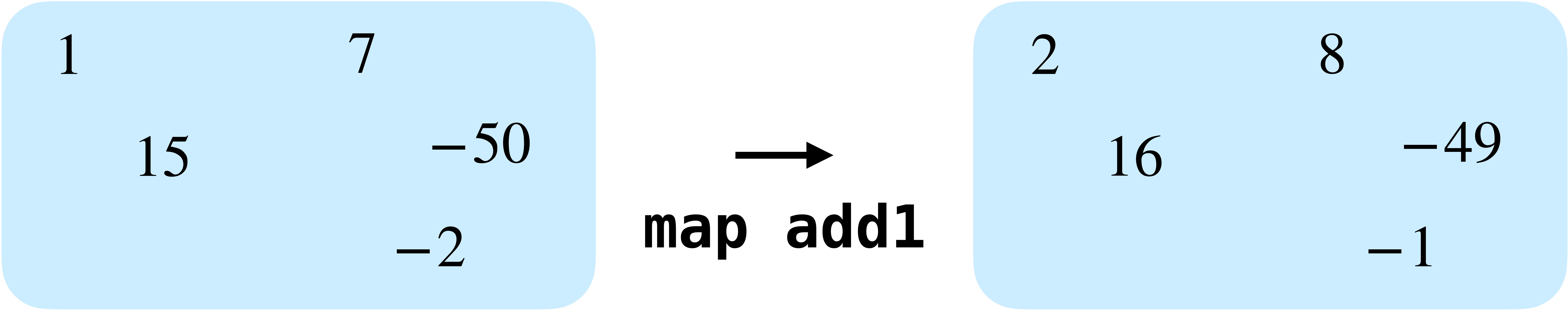
You may assume the list is nonempty, but as a challenge, try to write the function so that it returns an ***option.***

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.**

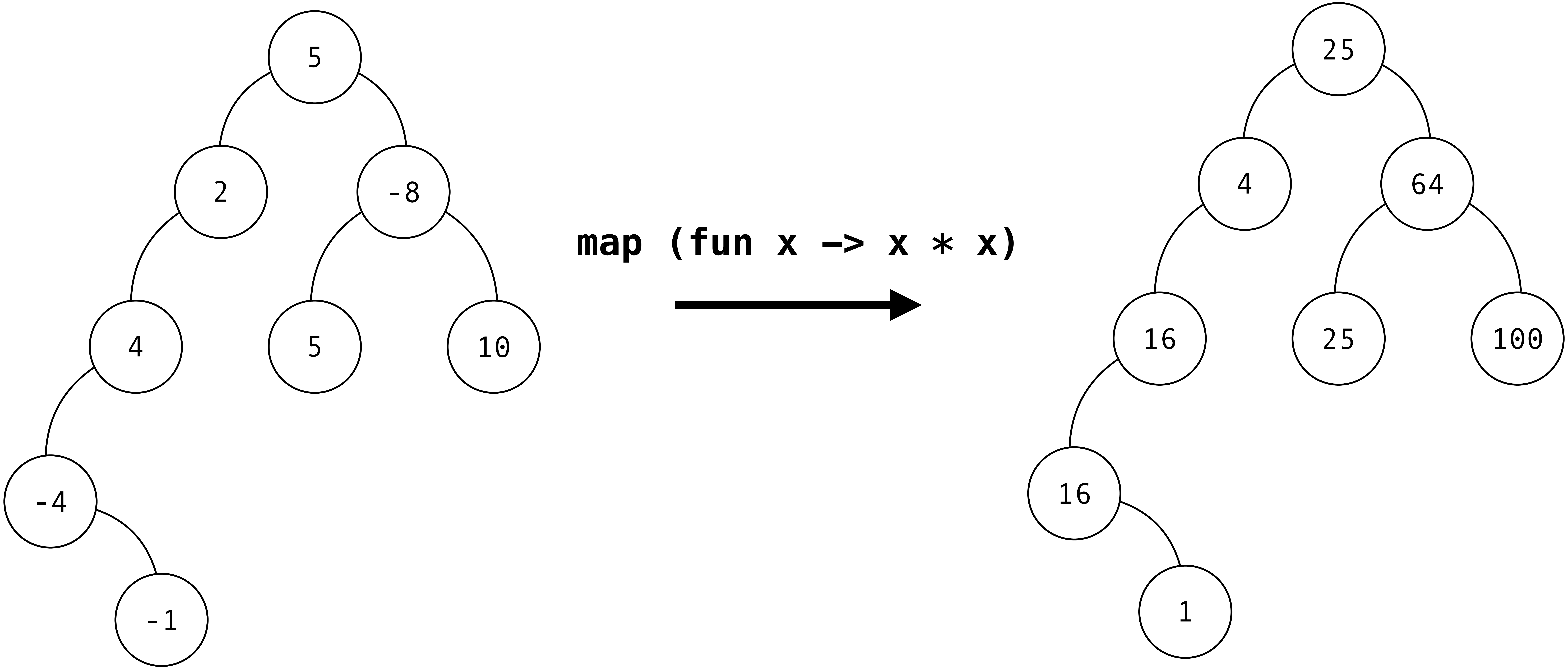


Example: Trees

```
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
```

*Keep the tree structure but recursively update the values with **f**.*

The Picture



Example: 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**.*

Example: Results

```
let map f ra =
  let rec go ra =
    match ra with
    | Error e -> Error e
    | Ok a -> Ok (f a)
  in go ra
```

*On **Error e**, leave the **Error e**.*

*On **Ok a**, apply **f** to **a**.*

Working with Options

```
let mkMatrix (vals : 'a list list) : 'a matrix option = ...
let transpose (mx : 'a matrix) : 'a matrix = ...
let vals = ...

let a = Option.map transpose (mkMatrix vals)
```

Working with Options

```
let mkMatrix (vals : 'a list list) : 'a matrix option = ...
let transpose (mx : 'a matrix) : 'a matrix = ...
let vals = ...

let a = Option.map transpose (mkMatrix vals)
```

Map allows us to "lift" non-option functions to option functions.

Working with Options

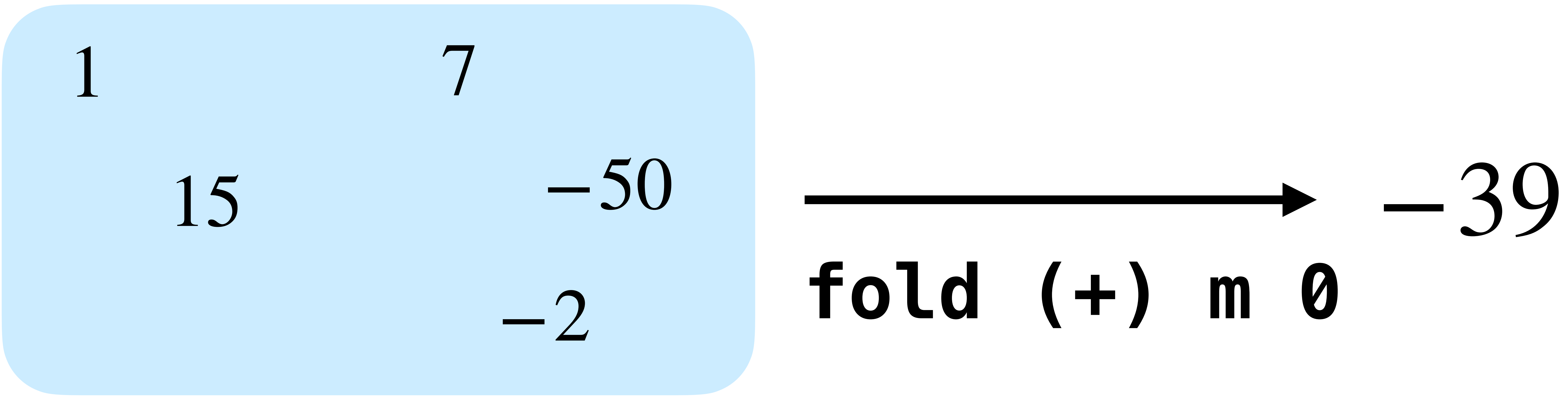
```
let mkMatrix (vals : 'a list list) : 'a matrix option = ...
let transpose (mx : 'a matrix) : 'a matrix = ...
let vals = ...

let a = Option.map transpose (mkMatrix vals)
```

Map allows us to "lift" non-option functions to option functions.

We can avoid pattern matching explicitly on options if we want to.

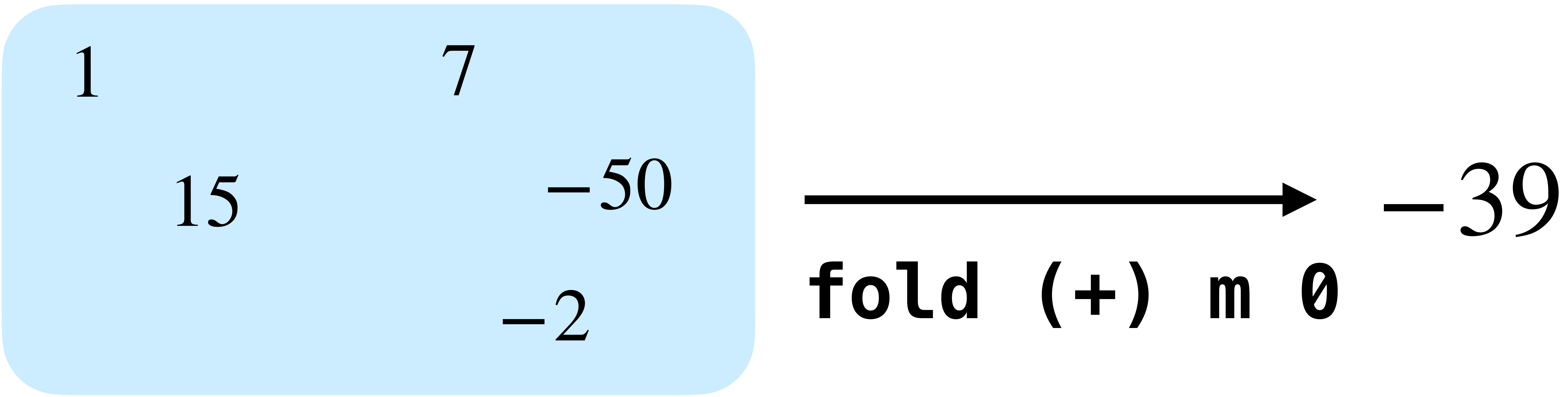
Foldable Data

There are also a lot of data types which hold uniform data that we might want to fold over.



Foldable Data

There are also a lot of data types which hold uniform data that we might want to fold over.



We have to deal with associativity and the order that elements are processed.

Example: Trees

```
let fold_left 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

Example: Trees

```
let fold_left 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.

Example: Trees

```
let fold_left 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.

Example: Trees

```
let fold_left 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
```

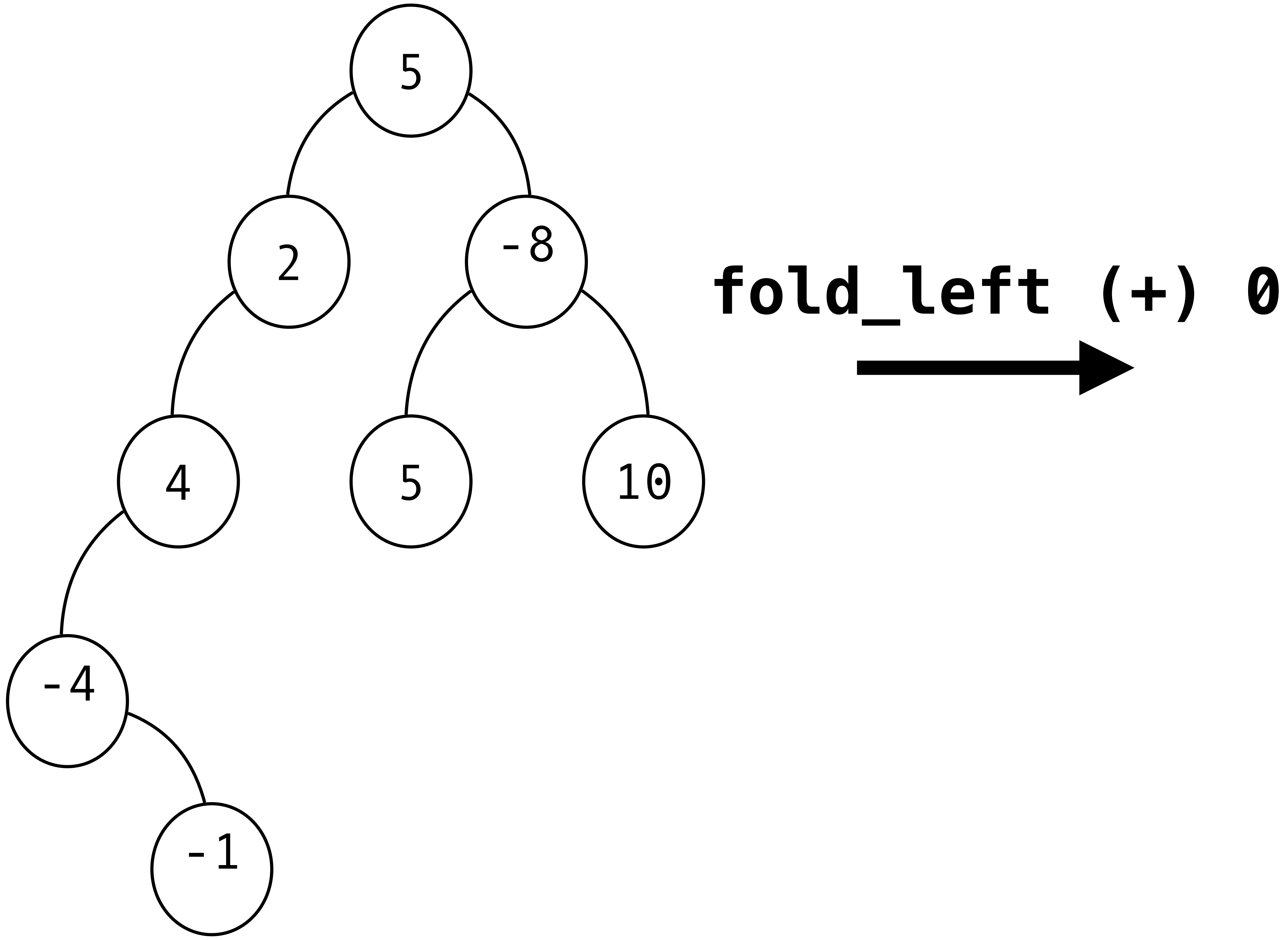
Not tail-recursive

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

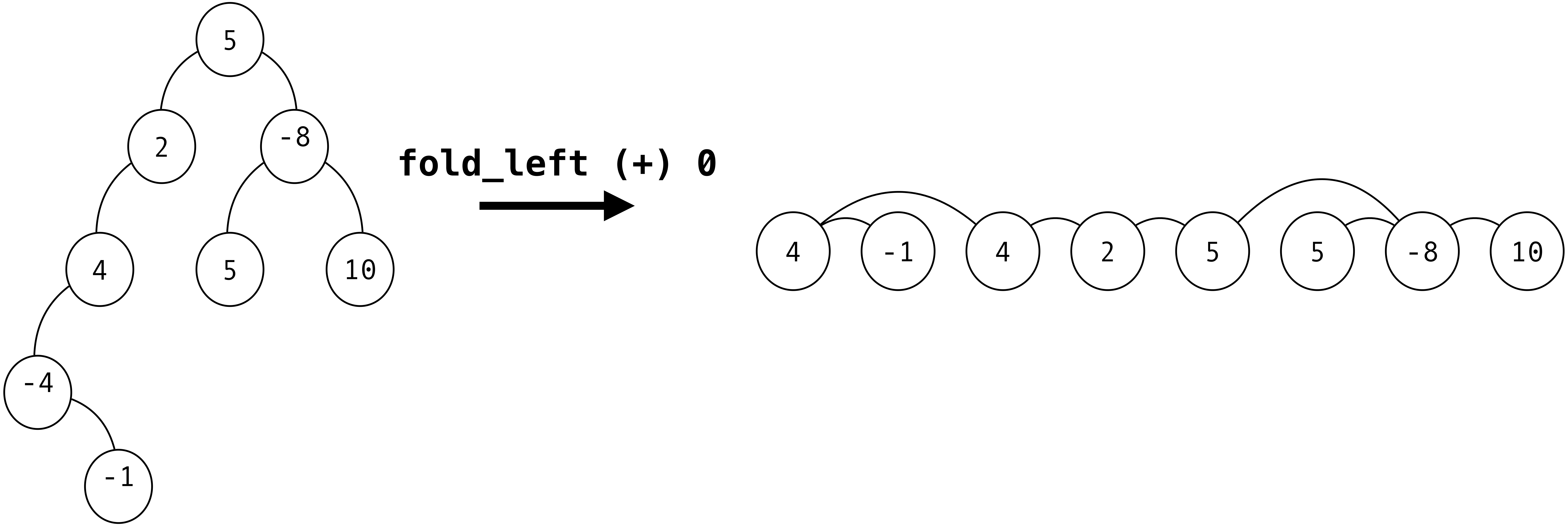
It is equivalent to "flattening" the tree into a list, and then folding that list.

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

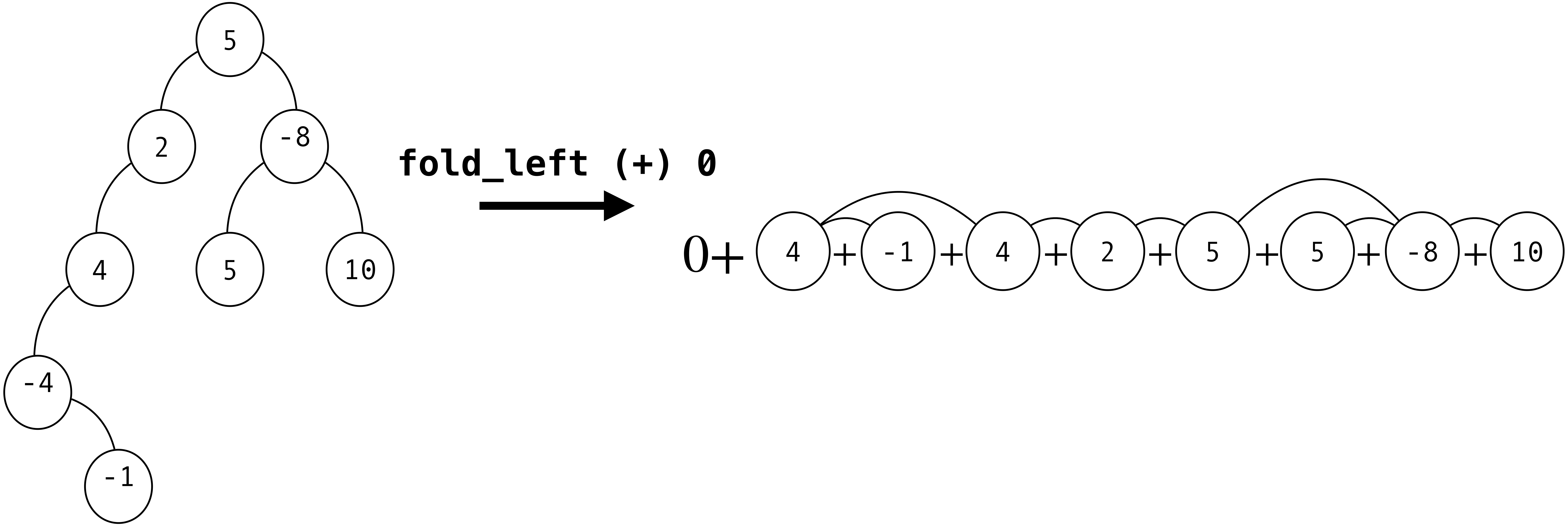
The Picture



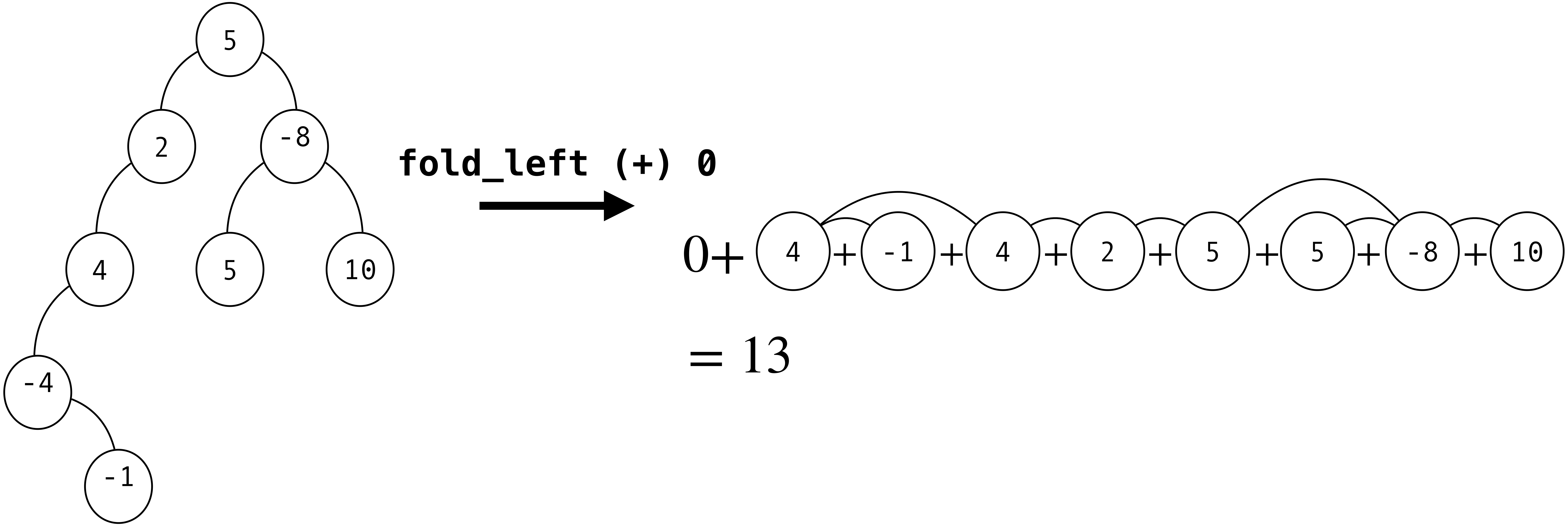
The Picture



The Picture



The Picture



Fold Right for Trees

```
let rec rev t =
  match t with
  | Leaf -> Leaf
  | Node (x, l, r) -> Node (x, rev r, rev l)

let fold_right op t base =
  fold_left (fun x y -> op y x) base (rev t)

let inorder t = fold_right (fun x xs -> x :: xs) t []
```

We can use the same trick to get **fold_right** from fold left.

Fold Right for Trees

```
let rec rev t =
  match t with
  | Leaf -> Leaf
  | Node (x, l, r) -> Node (x, rev r, rev l)

let fold_right op t base =
  fold_left (fun x y -> op y x) base (rev t)
                 "reverse" the operator       "reverse" the tree

let inorder t = fold_right (fun x xs -> x :: xs) t []
```

We can use the same trick to get **fold_right** from fold left.

Example: Options

```
let fold f base am =
  let rec go am =
    match am with
    | None -> base
    | Some x -> f base x
  in go am

(* Example based on Option.value *)
let value def ma = fold (fun _ x -> x) def ma
```

This may seem silly, but it allows us to perform a computation on the value inside an option, but have **base** as a "back-up plan".

Understanding Check

*Write a **map** function for the **concatlist**s from the last assignment.*

*Write a **fold_left** function for **concatlist**s.*

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.