Higher Order Programming: Folds

Concepts of Programming Languages
Lecture 8

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: 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**

They can be:

Recall: Higher-Order Programming

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

They can be:

1. <u>returned</u> by another function

Recall: Higher-Order Programming

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

They can be:

1. <u>returned</u> by another function
2. given names with <u>let-definitions</u>

Recall: Higher-Order Programming

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

They can be:

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

Recall: Higher-Order Programming

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

They can be:

1. <u>returned</u> by another function
2. given names with <u>let-definitions</u>
3. <u>passed as arguments</u> to another function

Note. Types are *not* first-class values

Recall: Higher-Order Programming

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

They can be:

1. returned by another function
2. given names with let-definitions
3. 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
```

This is *very* interesting in OCaml...

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 is *very* interesting in OCaml...

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

Recall: 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)
```

Some functions cannot be polymorphic

But can we still abstract the core functionality?

Recall: 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)
```

Some functions cannot be polymorphic

But can we still abstract the core functionality?

Recall: Simple Example

```
let rec accum 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

Recall: Simple Example

```
let rec accum 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

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 there is 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 there is nothing to do*
- *If the list is nonempty, we apply f to its first element, and recurse*

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 there is 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 there is nothing to do*
- *If the first element satisfies our predicate we keep 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 there is nothing to do*
- *If the first element satisfies our predicate we keep it and recurse*
- *Otherwise, we drop it and recurse*

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

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 rec. call

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

base rec. call

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

base rec. call

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

base rec. call

A Couple Functions

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

base rec. call combine

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

base rec. call combine

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

base rec. call combine

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

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

Fold as Specialized Pattern Matching

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

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
```

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] :: [])))))))
```

```
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 :: [])))))))
```

$\downarrow$ `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

```
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

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

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

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

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

Understanding Check

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

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

append (1 :: 2 :: 3 :: []) (4 :: 5 :: 6 :: [])

base

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

demo

(tail recursive fold attempt)

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

```
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

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

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_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
```

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 $x \mathbin{-_r} y := y - x$, 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**

General Rules for Folds

» For **associative** operations, use **fold_left**

» The types are difficult to remember, let the compiler remind you

General Rules for Folds

- For **associative** operations, use **fold_left**
- The types are difficult to remember, let the compiler remind you
- Don't use folds for everything, but also don't use pattern matching for everything. *Think about the use case*

Understanding Check

let sort le l =
List.fold_left
op (fun acc x ->
insert le x acc)
[ ]
l

```
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** fl op [ ] [2;1;3] $\Longrightarrow$

fl op (op [ ] 2) [1;3] $\Longrightarrow$

fl op [2] [1;3] $\Longrightarrow$

fl op (op [2] 1) [3] $\Longrightarrow$

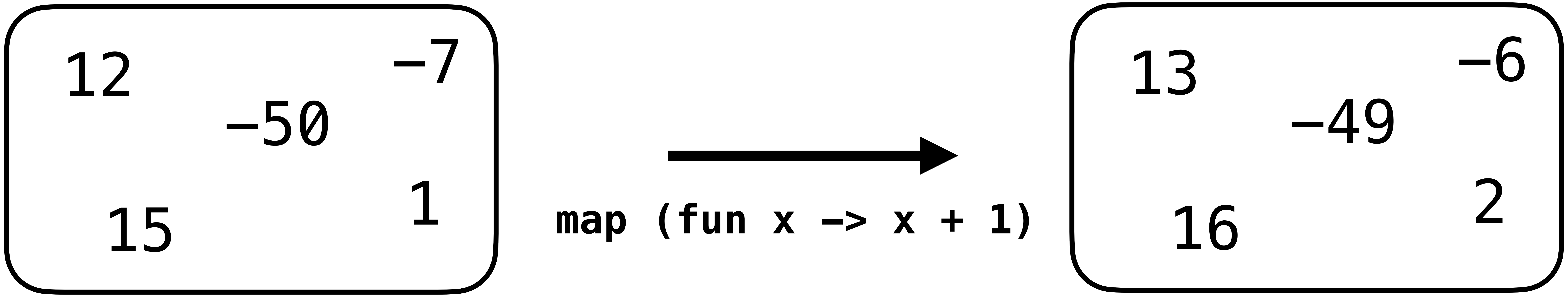
fl op [1;2] [3] $\Longrightarrow$

fl op (op [1;2] 3) [ ] $\Longrightarrow$

fl op [1;2;3] [ ] $\Longrightarrow$ [1;2;3]

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**

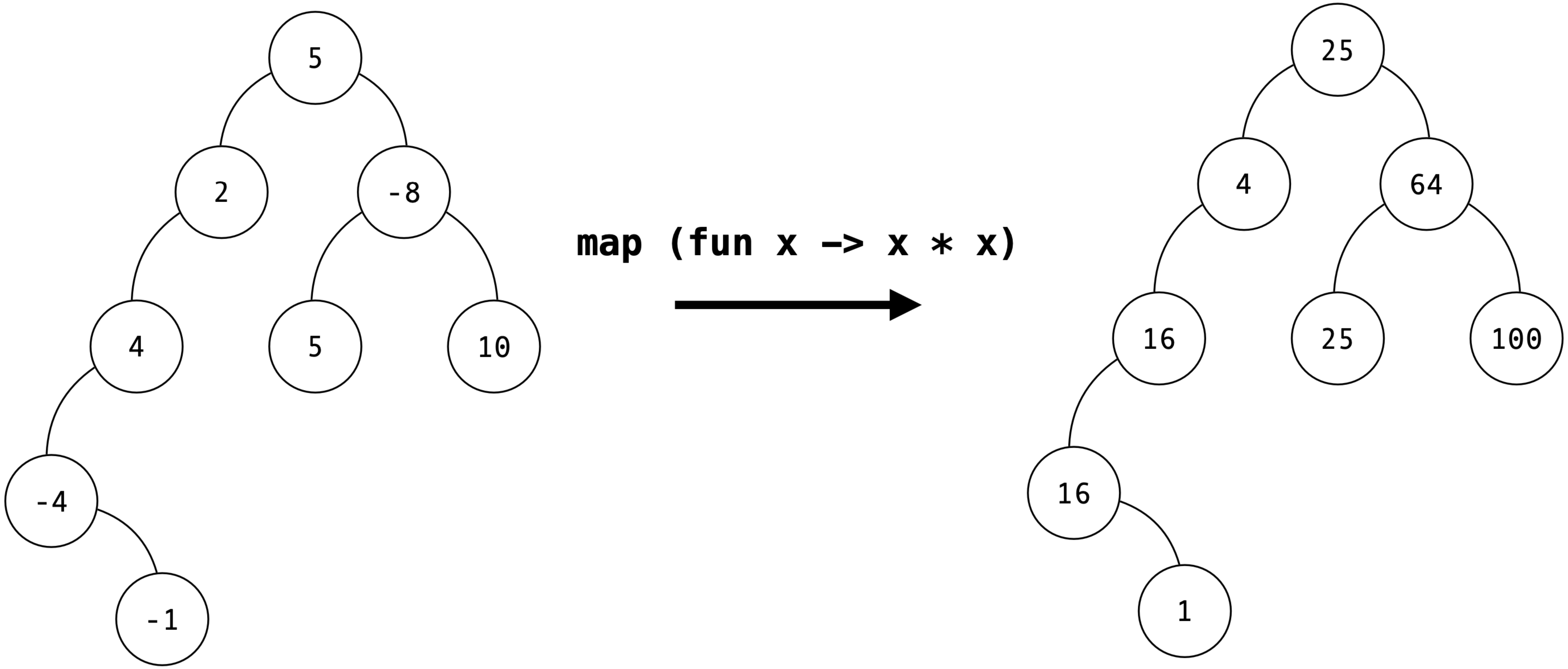
Trees

```
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***

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)
```

This is a very common pattern for working with options if we want to "keep computing" as long as the option still holds a value

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)
```

This is a very common pattern for working with options if we want to "keep computing" as long as the option still holds a value

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)
```

This is a very common pattern for working with options if we want to "keep computing" as long as the option still holds a value

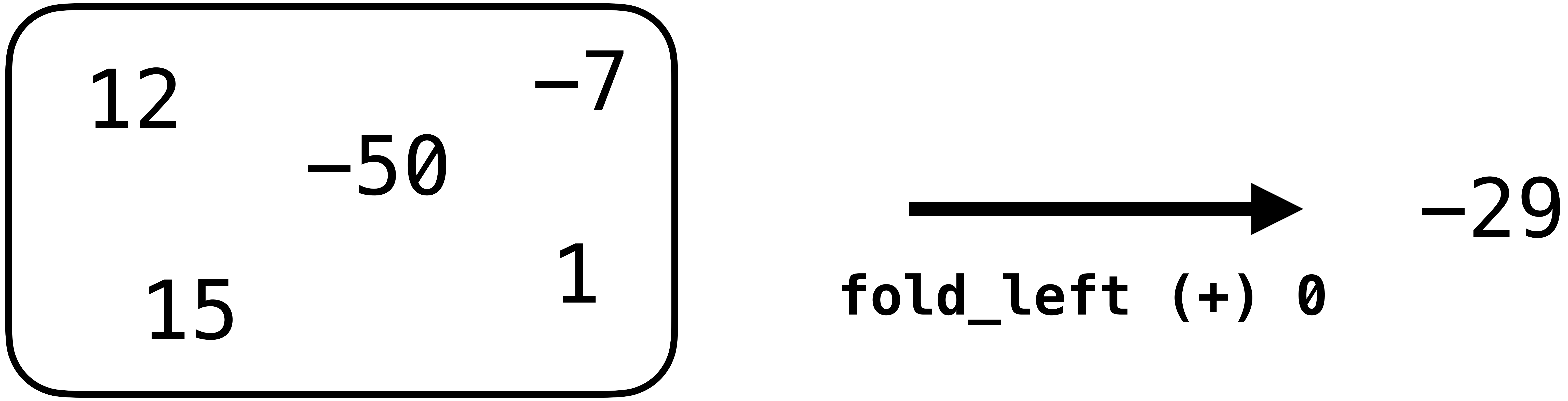
Map allows us to "lift" non-option functions to option functions

We can avoid pattern matching explicitly on options

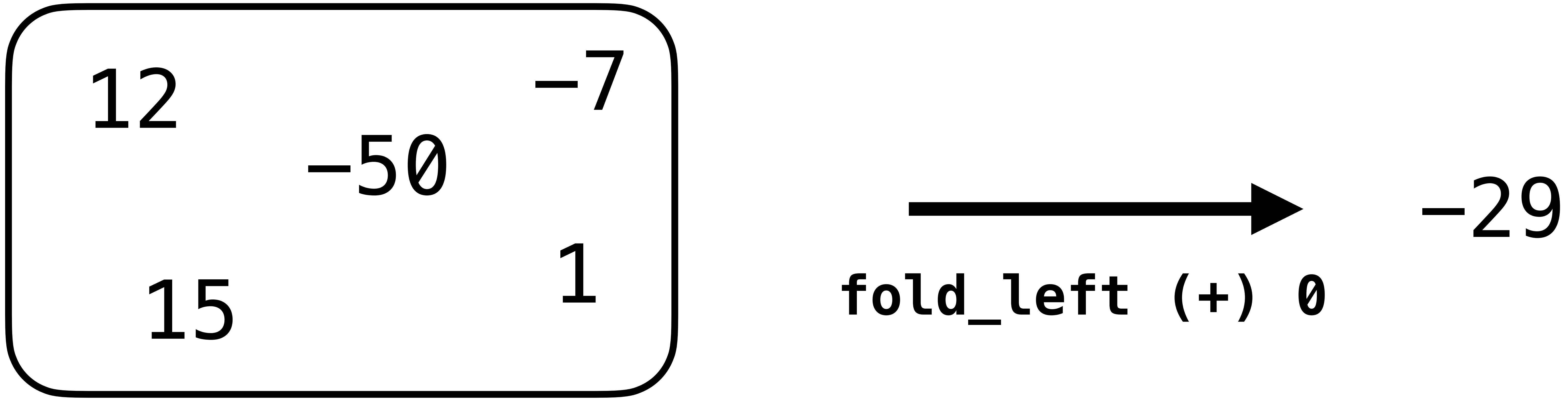
demo

(option mapping)

Foldable Data

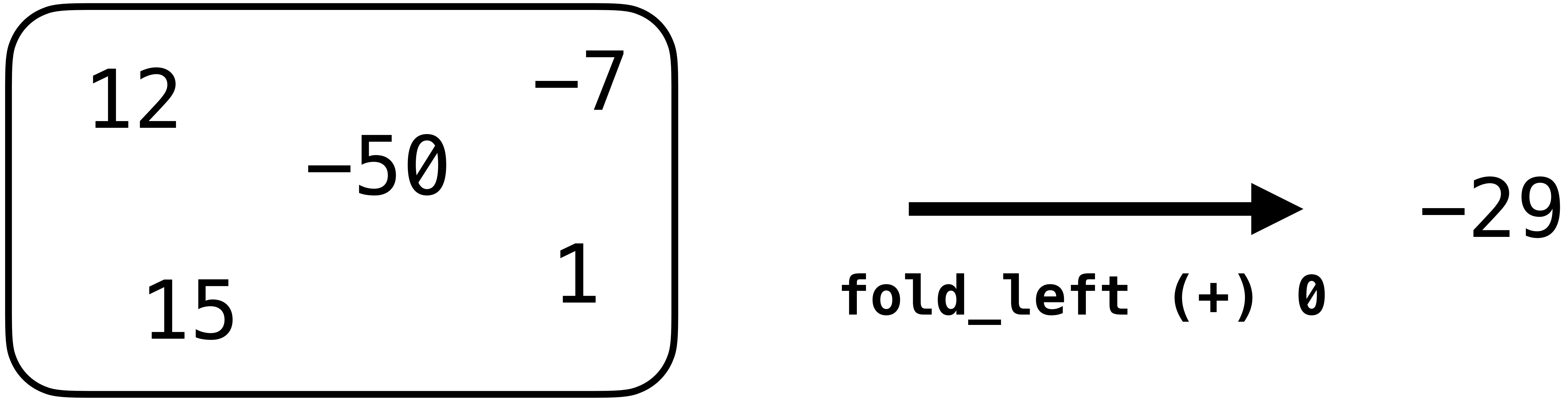


Foldable Data



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

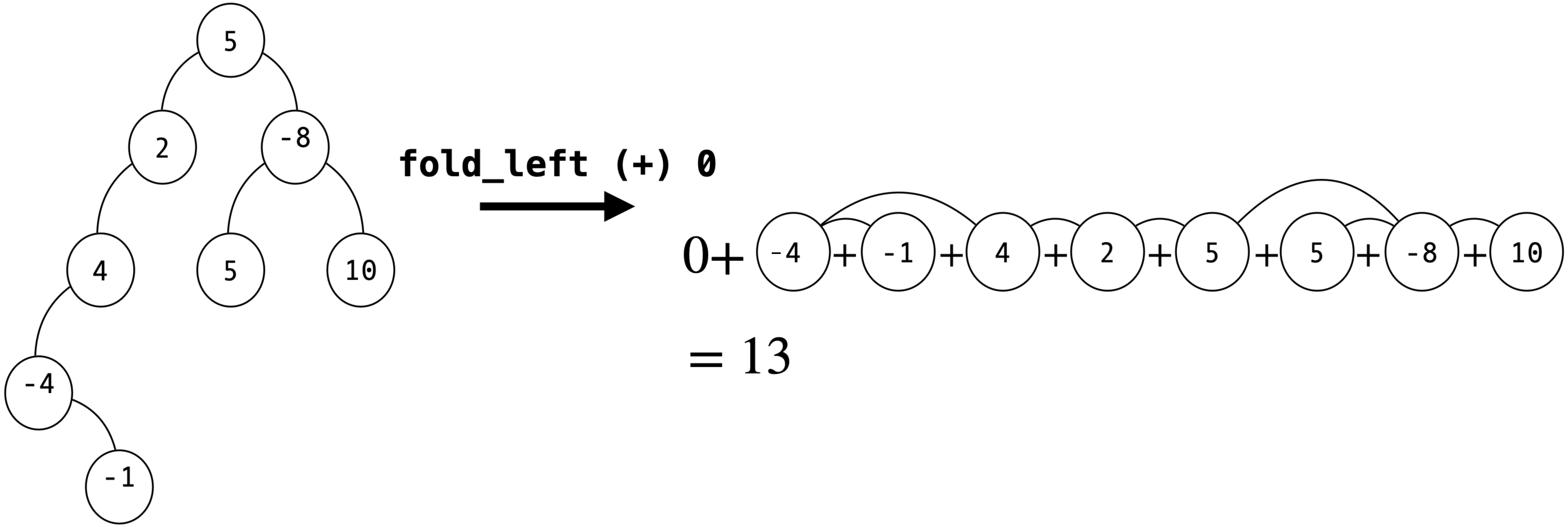
Foldable Data



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

Trees (The Picture)



Fold Left for 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
```

Fold Left for 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
```

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

Fold Left for 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
```

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

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

Fold Left for 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
```

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)

Fold Left for 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)

Understanding Check

*Implement **fold_left** for **ntree**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