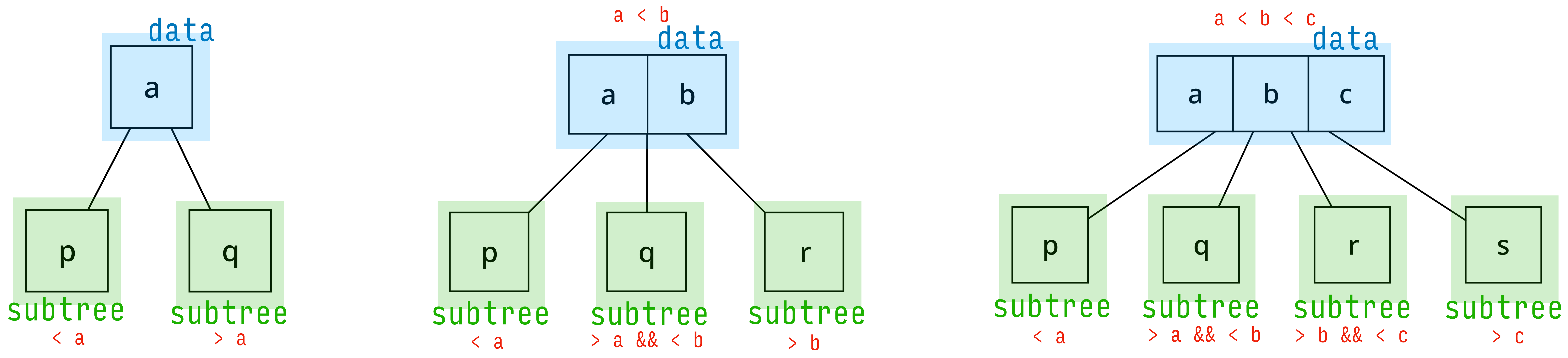
Higher-Order Programming

Concepts of Programming Languages

CAS CS 320

Practice Problem



A **2-3-4 tree** is a self-balancing tree structure with three possible kinds of nodes, shown above. Write an ADT to represent 2-3-4 trees

(If you have extra time, try implementing search for 2-3-4 trees)

Outline

- Introduce the notion of **higher-order functions** as a way to write cleaner, more general code
- Look at three common HOFs: **map**, **filter**, **fold**

Higher-Order Functions

Higher-Order Programming

Higher-Order Programming

In OCaml, functions can be:

Higher-Order Programming

In OCaml, functions can be:

1. **returned** by another function

Higher-Order Programming

In OCaml, functions can be:

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

Higher-Order Programming

In OCaml, functions can be:

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

Aside: Robin Popplestone

"He started a PhD at Manchester University before moving to Leeds University. His project was to develop a program for automated theorem proving, but he got caught up in **using the university computer to design a boat.** He built the boat and set sail for the University of Edinburgh, where he had been offered a research position. A storm hit while crossing the North Sea, and **the boat sank.** A widely believed story about Popplestone was that he never completed his PhD in mathematics because he **lost his thesis manuscript in the boat,** although Popplestone refused to corroborate this."

https://en.wikipedia.org/wiki/Robin_Popplestone

Functions as Return Values

```
# let f x y = x + y;;
val f : int -> int -> int = <fun>
# f 2;;
- : int -> int = <fun>
```

This isn't that interesting in OCaml because functions are **Curried**

Functions as Named Values

```
let f x y = x + y
```

is shorthand for...

```
let f = fun x -> fun y -> x + y
```

This also isn't that interesting because when we **let-define** *any* function, we're giving a anonymous function value a name

Functions as Named Values

```
let f x y = x + y
```

is shorthand for...

anonymous function

```
let f = fun x -> fun y -> x + y
```

This also isn't that interesting because when we **let-define** *any* function, we're giving a anonymous function value a name

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

Higher-Order Functions Elsewhere

fun $f \to \dfrac{f(x)}{dx}$ **e.g.** $x^2 \mapsto 2x$

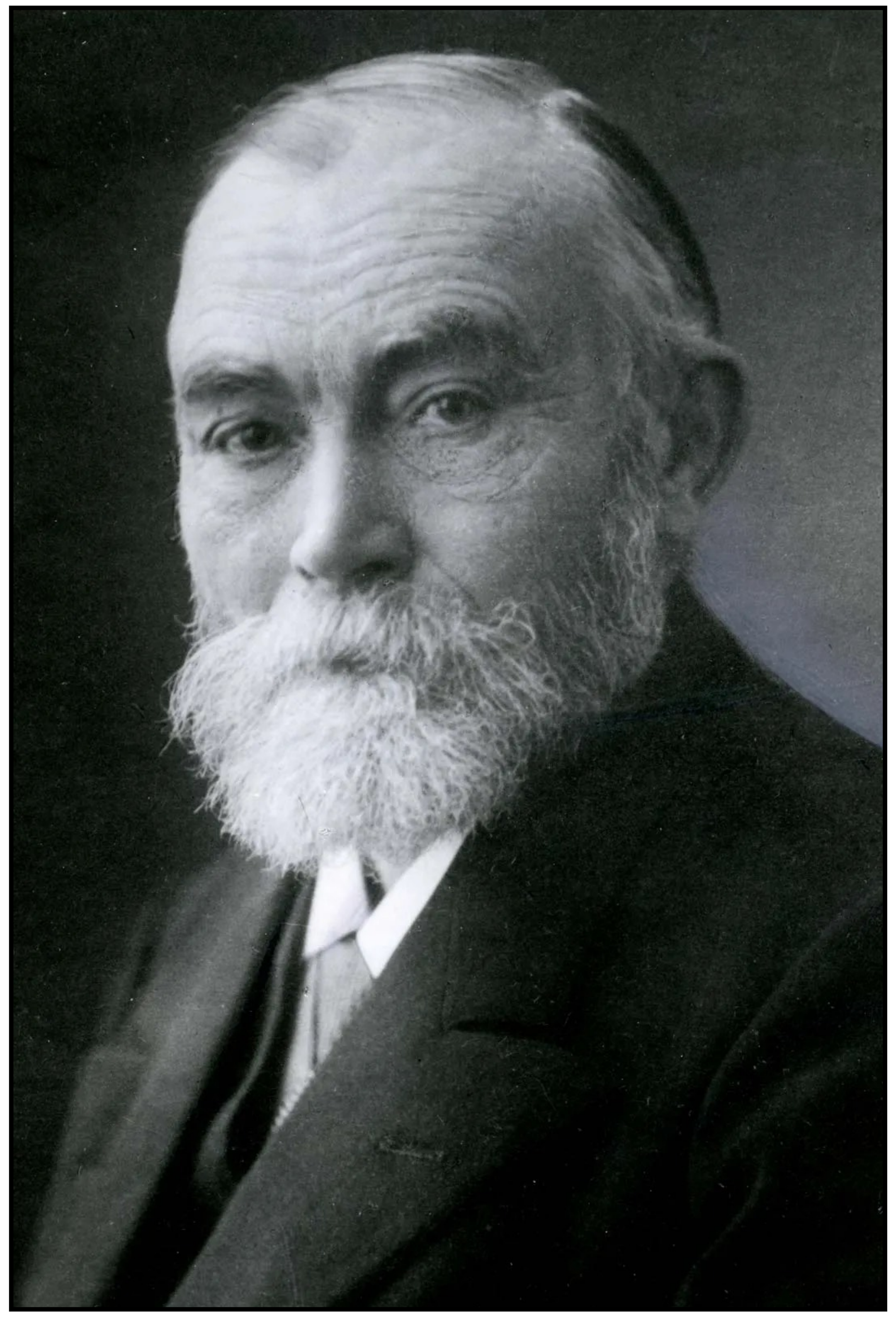
`We might think of the type of an` **`derivative`** `as`

$$(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$$

`because it takes one function and produces a new function`

Aside: What does "Higher-Order" Mean?

"Like things and functions are different, so are functions whose **arguments are functions** *radically different* from functions whose **arguments must be things**. I call the latter functions of first order, the former functions of second order."


Gottlob Frege

https://softwareengineering.stackexchange.com/questions/186035/who-first-coined-the-term-higher-order-function-and-or-first-class-citizen

First-Order Function Types

```
int -> string

() -> bool

bool * bool -> bool

'a -> 'a
```

Second-Order Function Types

```
(int -> string) -> int -> string

((() -> bool) -> bool

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

Third-Order Functions

```
(('a -> 'b) -> 'a) -> 'b

((() -> bool) -> bool) -> bool

((bool -> bool) -> bool) * bool -> bool
```

And so on…

```
1st: int
2nd: int -> int
3rd: (int -> int) -> int
4th: ((int -> int) -> int) -> int
5th: (((int -> int) -> int) -> int) -> int
6th: ((((int -> int) -> int) -> int) -> int) -> int
7th: (((((int -> int) -> int) -> int) -> int) -> int) -> int
8th: ((((((int -> int) -> int) -> int) -> int) -> int) -> int) -> int
⋮
```

The **higher-order** part comes from the fact that we can do this *ad infinitum*

(In practice, we rarely use higher than third-order or fourth-order functions)

The Abstraction Principle

The Abstraction Principle

The Abstraction Principle

One of the three virtues of a great programmer is *laziness*

The Abstraction Principle

One of the three virtues of a great programmer is *laziness*

The **abstraction principle** helps use be lazy

The Abstraction Principle

One of the three virtues of a great programmer is *laziness*

The **abstraction principle** helps use be lazy

When we write general programs, we *avoid rewriting programs* we've (pretty much) written before

Simple Example

```
let rec reverse (l : int list) : int list =
  match l with
  | [] -> []
  | x :: xs -> reverse xs @ [x]
```

Remember that **polymorphism** allows us to write general functions by being *agnostic* about types

*It doesn't matter if we're reversing an **int list** of **string list** or an **int list list**...*

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?

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?

demo

(accumulate)

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
```

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
```

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*

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

Another Example

```
let rec insert (x : 'a) (l : 'a list) : 'a list =
  match l with
  | [] -> [x]
  | y :: ys -> if x <= y then x :: y :: ys else y :: insert x ys

let rec sort (l : 'a list) : 'a list =
  match l with
  | [] -> []
  | x :: xs -> insert x (sort xs)
```

Sorting *is* polymorphic

But what if we want to sort in *reverse order*, or *only on a part of the data*?

demo

(sorting)

The Abstraction Principle

The Abstraction Principle

The abstraction principle comes from MacLennan's **Functional Programming: Theory and Practice**

The Abstraction Principle

The abstraction principle comes from MacLennan's **Functional Programming: Theory and Practice**

» Abstract out <u>core functionality</u>

The Abstraction Principle

The abstraction principle comes from MacLennan's **Functional Programming: Theory and Practice**

» Abstract out core functionality

» Use higher-order functions to parametrize by functionality specific to the problem

The Abstraction Principle

The abstraction principle comes from MacLennan's **Functional Programming: Theory and Practice**

» Abstract out core functionality

» Use higher-order functions to parametrize by functionality specific to the problem

» (Try to understand the algebra of programming)

Practice Problem

Implement the function

```
val negatives : int list -> int list
```

so that **negatives l** *is the list negative numbers appearing in* **l.**
Also implement the function

```
val gets : 'a -> ('a * 'b) list -> 'b list
```

so that **gets key l** *is the list of values* **v** *such that* **(key, v)** *is a member of* **l**

Write a single function that can be used to implement both

Three Common HOFs

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

Map

Example

```
type user = {
  name : string ;
  id : int ;
}

let capitalize = ...

let fix_usernames (us : user list)  =
  List.map (fun u -> { u with name = capitalize u.name }) us
```

`map` is used to apply a function to every element in a list (or other structure)

Definition of Map

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

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*

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*

Definition of Map

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

Is this tail recursive?

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

» *If the list is nonempty, we apply f to its first element, and recurse*

Tail-Recursive Map

```
let rec map_t f l =
  let rec go l acc =
    match l with
    | [] -> List.rev acc
    | x :: xs -> go xs (f x :: acc)
  in go l []
```

Tail-Recursive Map

```
let rec map_t f l =
  let rec go l acc =
    match l with
    | [] -> List.rev acc
    | x :: xs -> go xs (f x :: acc)
  in go l []
```

For a tail-recursive version we can build the list in reverse in acc and then *reverse it at the end*

Tail-Recursive Map

```
let rec map_t f l =
  let rec go l acc =
    match l with
    | [] -> List.rev acc
    | x :: xs -> go xs (f x :: acc)
  in go l []
```

For a tail-recursive version we can build the list in reverse in acc and then *reverse it at the end*

This may seem inefficient, but it's just a *constant factor* slower

demo

(normalize)

Practice Problem

Implement then function

```
  val pointwise_max : ('a -> int) -> ('a -> int)
                          -> 'a list -> 'a list
```

so that ***pointwise_max f g l*** *is* ***l*** *but with* ***f*** *or* ***g*** *applied to each element, whichever gives the larger value*

Filter

Example

```
type user = {
  name : string ;
  id : int ;
  num_likes : int ;
}

let popular (us : user list) (cap : int) =
  List.filter (fun u -> u.num_likes > cap) us
```

`filter` is used to do grab all elements in a list which *satisfy a given property*

Predicates

Def. A **Boolean predicate** on **'a** is a function of type **'a -> bool**

A predicate is a function which defines a *property*

Examples:

```
let even n = n mod 2 = 0
let even_length l = even (List.length l)
```

Definition of Filter

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

Definition of 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*

Definition of 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*

Definition of 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*

Definition of Filter

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

Is this tail recursive?

- *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*

Tail-Recursive Definition of Filter

```
let filter_tail p =
  let rec go acc l =
    match l with
    | [] -> List.rev acc
    | x :: xs -> go ((if p x then [x] else []) @ acc) xs
  in go []
```

As with map, we have to reverse the output before returning it

demo

(primes)

Question

```
  let h p q = List.filter (fun i -> p i && q i)
```

What does the above function do?

Folds

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 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
```

Fold as Specialized Pattern Matching

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

We get a one-liner!

The point: folds can "iterate" 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

$$\begin{tikzcd}
\texttt{1 :: (2 :: (3 :: (4 :: (5 :: (6 :: (7 :: [])))))))} \arrow[d, "\texttt{prod = fold\_right (*) l 1}"] \\
\texttt{1 * (2 * (3 * (4 * (5 * (6 * (7 * 1)))))))}
\end{tikzcd}$$

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

The Picture

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

$\Big\downarrow$ `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 :: [])))))))
                       ↓ 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

Practice Problem

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

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

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

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

Ex. 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

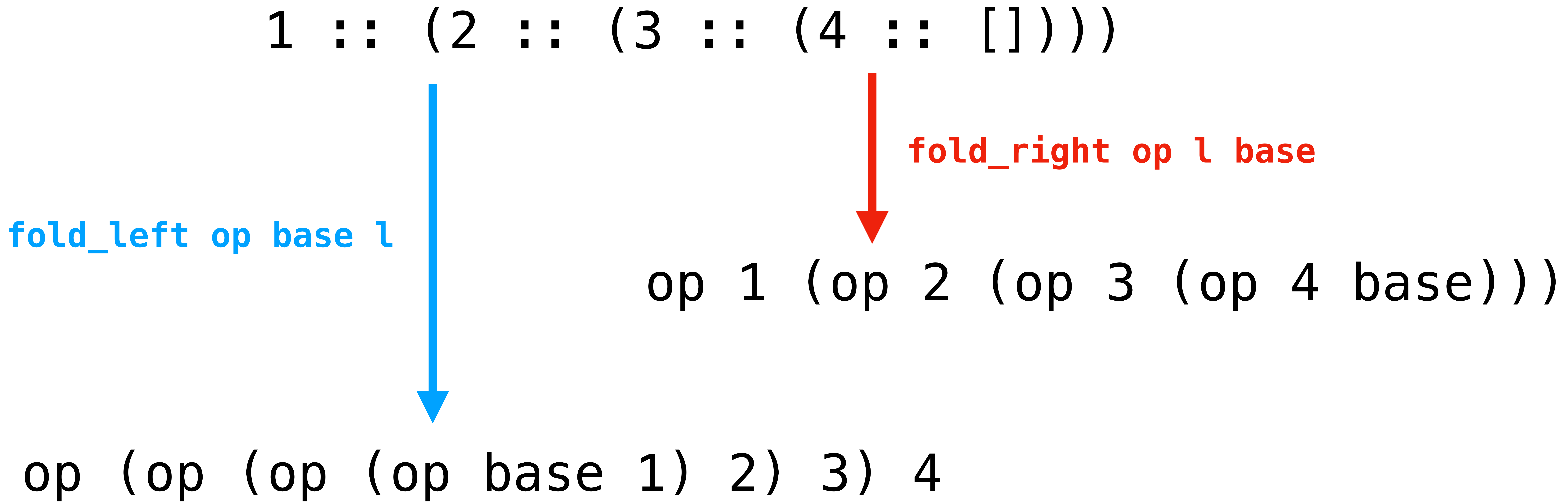
note the order of args.

```
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
```

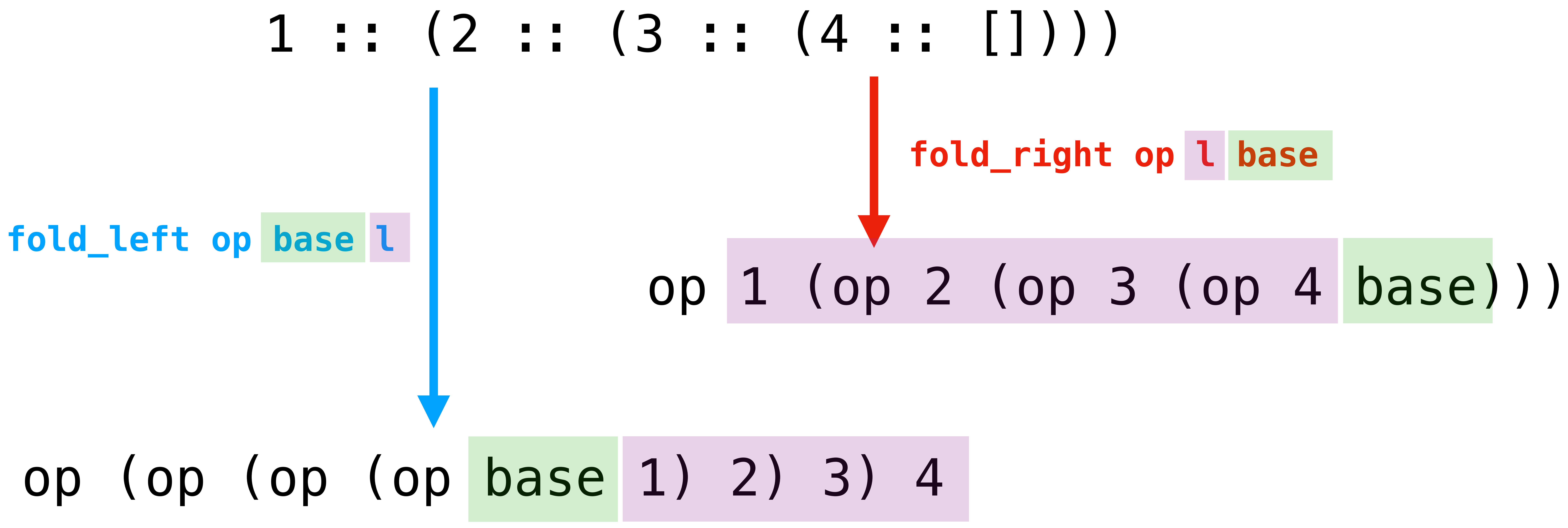
Folding left is just our incorrect tail recursive right folding (with a change in the order of arguments)

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** -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*

demo

(normalize)

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
```

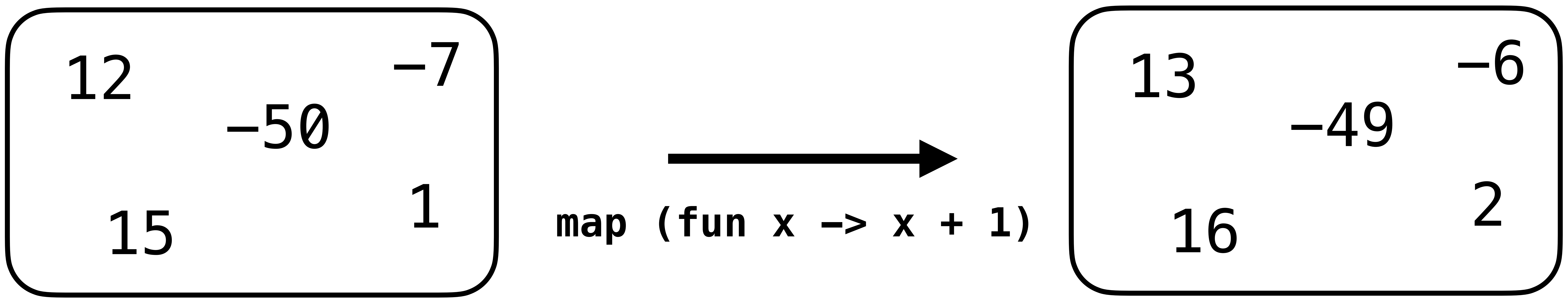
In terms of **fold_left** implement the function

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**

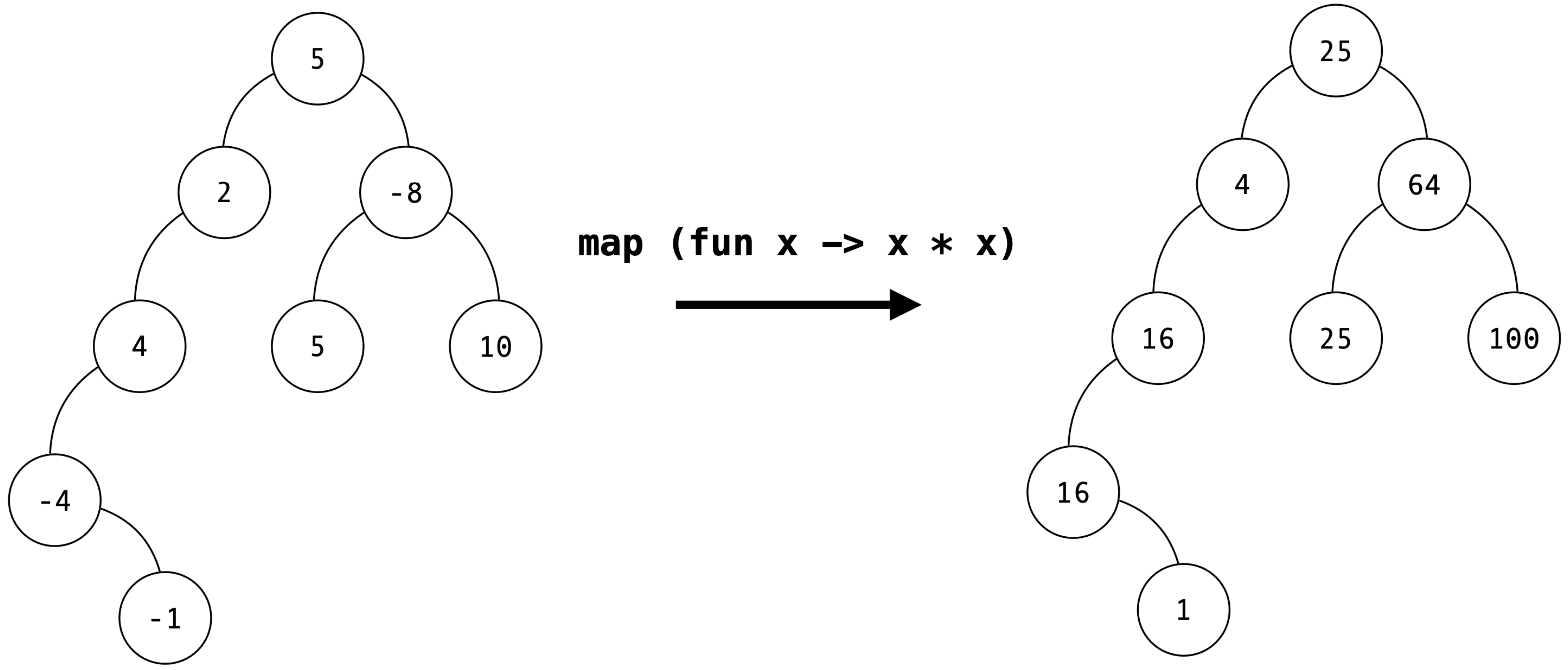
Trees

```
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



```
map (fun x -> x * x)
```

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

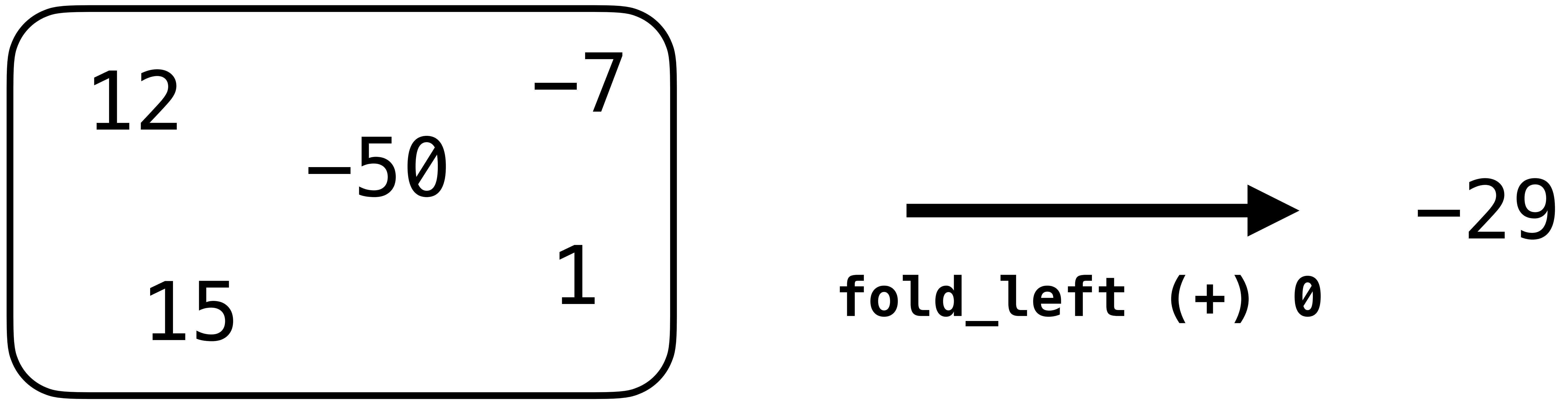
Map allows us to "lift" non-option functions to option functions

We can avoid pattern matching explicitly on options

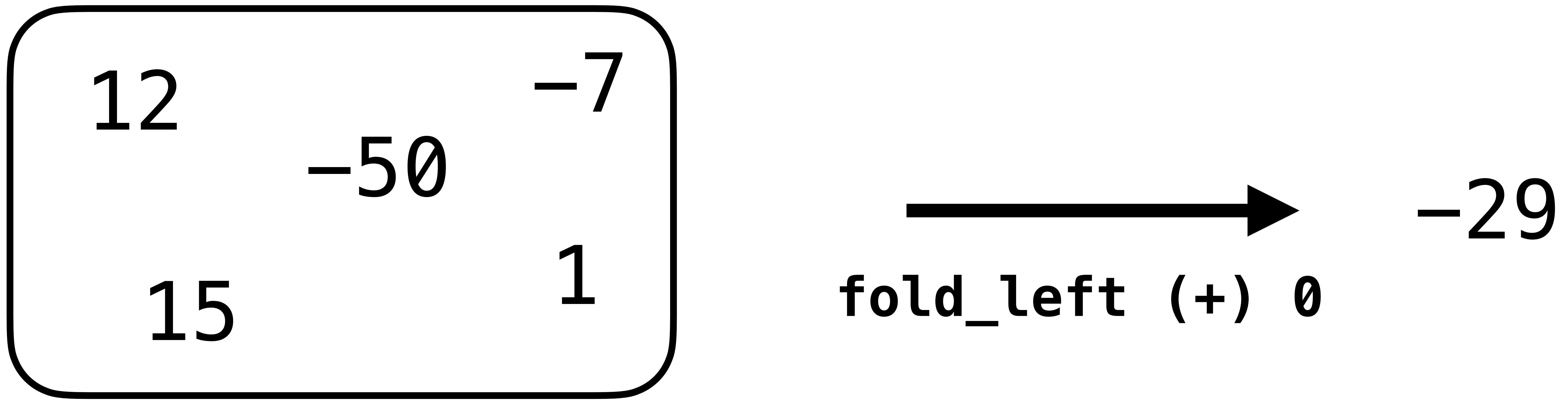
demo

(option mapping)

Foldable Data

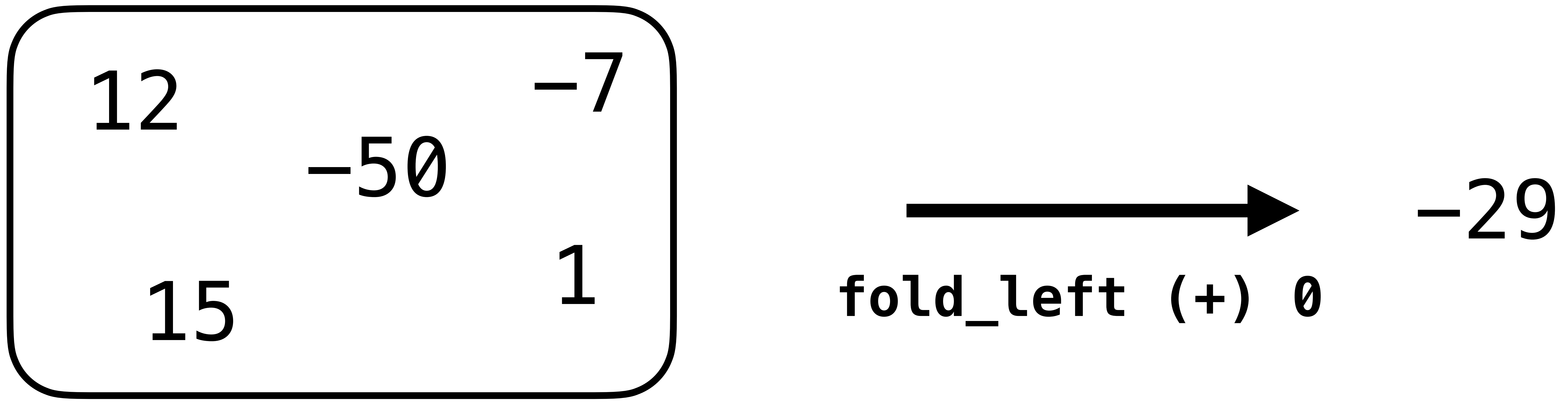


Foldable Data



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

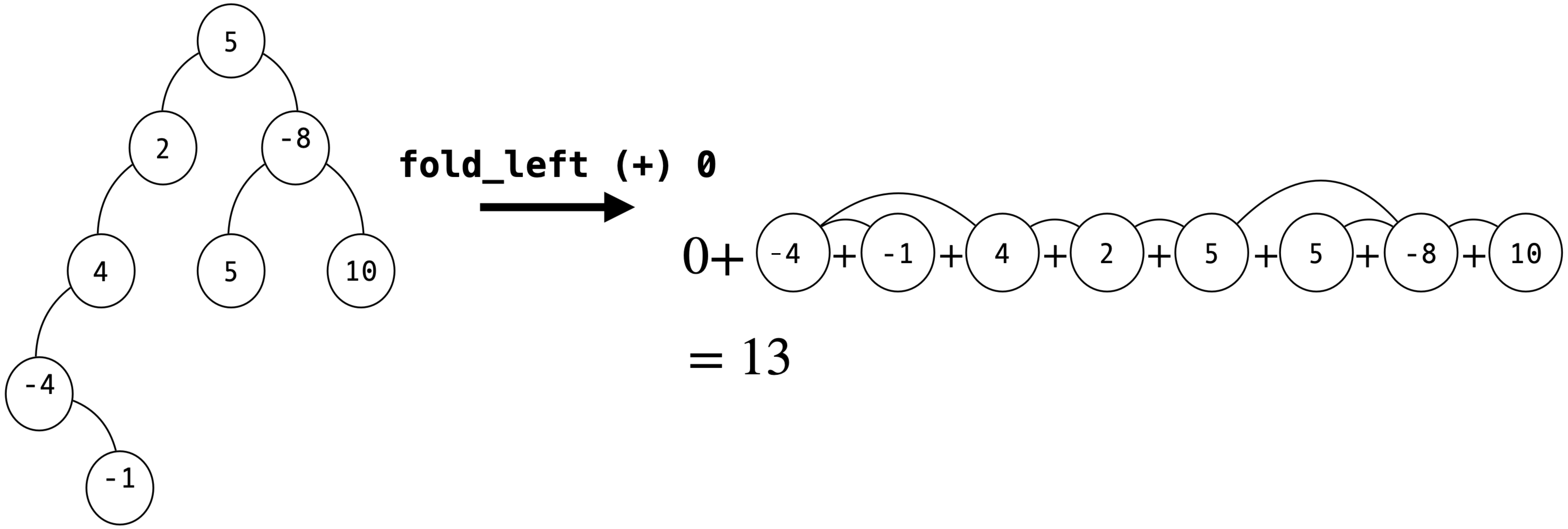
Foldable Data



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)

Summary

Higher-order function allow for better **abstraction** because we can **parameterize** functions by other functions

map and **filter** and **fold** very common patterns which can be used to write clean and simple code

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