Functional Programming

2022-2023

Sjaak Smetsers

Higher-order functions
Lecture 5

Outline

- Functions as first-class citizens

- Functions as arguments

- Functions as results

- Folds and unfolds

- Summary

Functions as first-class citizens

In programming language design, a first-class citizen (also type, object, entity, or value) in a given programming language is an entity which supports all the operations generally available to other entities. These operations typically include being passed as an argument, returned from a function, and assigned to a variable.
        source: **https://en.wikipedia.org/wiki/First-class_citizen**

- functional programming concerns functions (of course!)
- functions are *first-class citizens* of the language
  - functions have all the rights of other types:
    - may be passed as arguments
    - may be returned as results
    - may be stored in data structures
- functions that manipulate functions are *higher-order*

Functions as arguments

- we have already seen examples of higher-order operators encapsulating *patterns of computation*, aka *program schemes*:

```
map  :: (a→b)→[a]→[b]
map f []     = []
map f (x:xs) = f x : map f xs


filter :: (a→Bool)→[a]→[a]
filter p []    = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

- each is a parameterizable program scheme
- parameterization improves modularity, and hence understanding, modification, and reuse

Functions as arguments — continued

- quickSort relies on predefined ordering (< and ≤):

```
quickSort :: Ord a ⟹ [a] → [a]
quickSort [ ]      = [ ]
quickSort (x : xs) = quickSort littles ++ [x] ++ quickSort bigs
   where littles = [a | a ← xs, a < x]
         bigs    = [a | a ← xs, x ≤ a]
```

Functions as arguments — continued

- `quickSort` relies on predefined ordering (`<` and `≤`):

```
quickSort :: Ord a ⟹ [a] → [a]
quickSort [ ]      = [ ]
quickSort (x : xs) = quickSort littles ++ [x] ++ quickSort bigs
  where littles = [a | a ← xs, a < x]
        bigs    = [a | a ← xs, x ≤ a]
```

- abstract away from ordering

```
quickSortBy :: (a → a → Bool) → [a] → [a]
quickSortBy cmp [ ]    = [ ]
quickSortBy cmp (x:xs) = quickSortBy cmp lefts ++[x]++ quickSortBy cmp rights

  where lefts  = [a | a ← xs, not (x `cmp` a)]
        rights = [a | a ← xs, x `cmp` a]
```

Functions as arguments — continued

- **quickSort** relies on predefined ordering (**<** and **≤**):

```haskell
quickSort :: Ord a ⇒ [a] → [a]
quickSort [ ]      = [ ]
quickSort (x : xs) = quickSort littles ++ [x] ++ quickSort bigs
  where littles = [a | a ← xs, a < x]
        bigs    = [a | a ← xs, x ≤ a]
```

- abstract away from ordering

```haskell
quickSortBy :: (a → a → Bool) → [a] → [a]
quickSortBy cmp [ ]    = [ ]
quickSortBy cmp (x:xs) = quickSortBy cmp lefts ++[x]++ quickSortBy cmp rights
  where lefts  = [a | a ← xs, not (x `cmp` a)]
        rights = [a | a ← xs, x `cmp` a]
```

- more flexible/general e.g. **quickSortBy (>)**

cf. sorting in Java

- Sorting specified list according to the natural ordering of its elements

static `<T` **extends** `Comparable<T>>` `void` `sort`(`List<T>` list)

- Sorting the specified list according to the order induced by the specified comparator..

static `<T>` `void` `sort`(`List<T>` list, `Comparator<T>` c)

- Here:

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2)
}
```

Functions as results

- functions may also be returned as results

```
data Op = Add | Sub | Mul | Div
op :: Op → (Integer → Integer → Integer)
op Add = (+)
op Sub = (-)
op Mul = (*)
op Div = div
```

- partial application

- currying

- function composition

Partial application

- consider `add' x y = x + y`

- type `Integer` $\longrightarrow$ `Integer` $\longrightarrow$ `Integer`; takes two integers and returns an integer (e.g. `add' 3 4 = 7`)

- another view: type `Integer` $\longrightarrow$ (`Integer` $\longrightarrow$ `Integer`) ($\longrightarrow$ associates to the right); takes a single Integer and returns an `Integer` $\longrightarrow$ `Integer` function
  - e.g. `add' 3` is the function that adds three to its argument

- need not apply function to all its arguments at once: *partial application*; result will then be a function, awaiting remaining arguments (e.g. `op`, `op Add`, `op Add 47`; the expression `op Add 47 11` is a full application)

- *sectioning* (`(3 +)`, `(+)`, `(< 4)`) is partial application of binary operators
  - what's the type of `(< 4)`?

Currying

- in Haskell, every function takes *exactly* one argument
- a function taking pair of arguments can be transformed into a function taking two successive arguments, and vice versa

```
add :: (Integer, Integer) → Integer
add (x, y) = x + y
add' :: Integer → Integer → Integer
add' x y = x + y
```

- **add'** is called the curried version of **add**
- named after logician Haskell B. Curry (like the language)
- in fact, curried functions are the norm in Haskell

Currying — continued

- transformations are implementable as higher-order operations

  curry :: ((a,b) → c) → (a → b → c)

  uncurry :: (a → b → c) → ((a,b) → c)

Currying — continued

- transformations are implementable as higher-order operations

  `curry` `::` `(`$(a,b) \rightarrow c$`)` $\rightarrow$ `(`$a \rightarrow b \rightarrow c$`)`

  `curry` `f` `a` `b` `=` `f` `(a,b)`

  `uncurry` `::` `(`$a \rightarrow b \rightarrow c$`)` $\rightarrow$ `(`$(a,b) \rightarrow c$`)`

Currying — continued

- transformations are implementable as higher-order operations
  curry :: ((a,b) → c) → (a → b → c)
  curry f a b = f (a,b)
  uncurry :: (a → b → c) → ((a,b) → c)
  uncurry f (a,b) = f a b
- e.g. add' = curry add
- a related higher-order operation: flip arguments of binary function
  - (later: reverse = foldl (flip (:)) [ ])

Currying — continued

- transformations are implementable as higher-order operations

  `curry :: ((a,b) → c) → (a → b → c)`

  `curry f a b = f (a,b)`

  `uncurry :: (a → b → c) → ((a,b) → c)`

  `uncurry f (a,b) = f a b`

- e.g. `add' = curry add`
- a related higher-order operation: flip arguments of binary function
  - (later: `reverse = foldl (flip (:)) [ ]`)

  `flip :: (a → b → c) → (b → a → c)`

Currying — continued

- transformations are implementable as higher-order operations

  ```
  curry :: ((a,b) → c) → (a → b → c)
  curry f a b = f (a,b)
  uncurry :: (a → b → c) → ((a,b) → c)
  uncurry f (a,b) = f a b
  ```

- e.g. `add' = curry add`
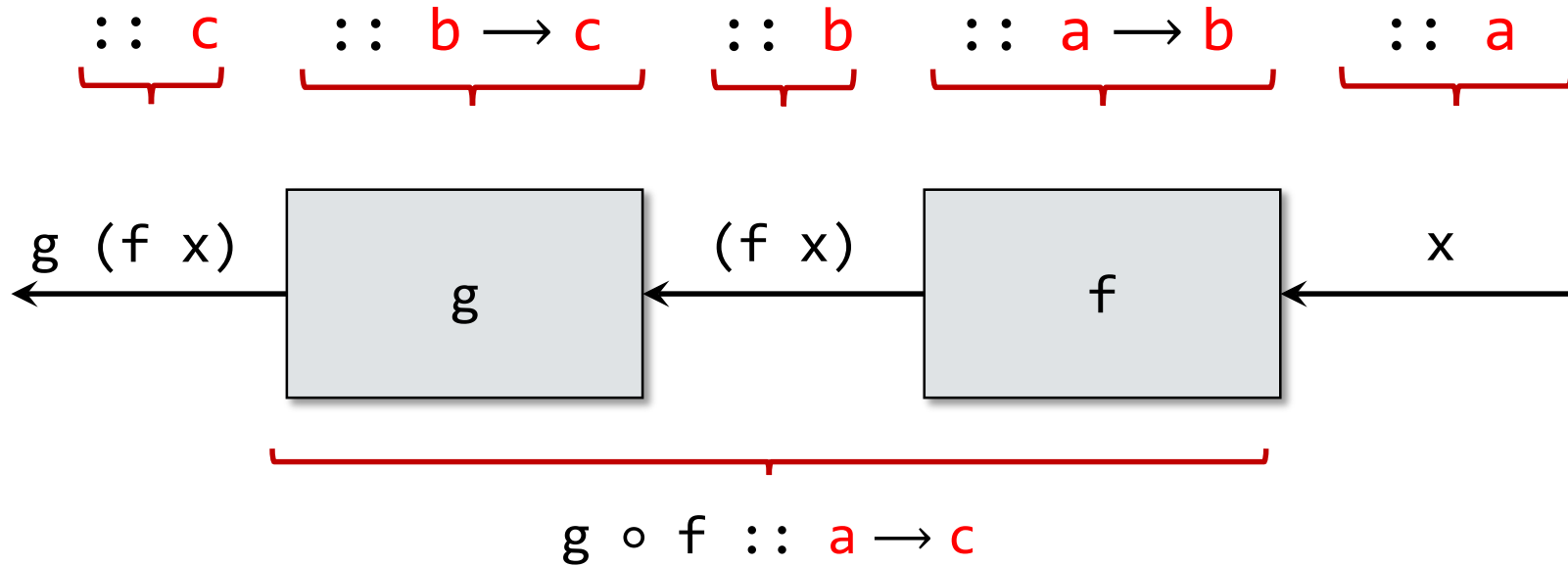- a related higher-order operation: flip arguments of binary function
  - (later: `reverse = foldl (flip (:)) [ ]`)

  ```
  flip :: (a → b → c) → (b → a → c)
  flip f b a = f a b
  ```

Function composition

:: c   :: b → c   :: b   :: a → b   :: a

g (f x)   (f x)   x

| g | | f |

g ∘ f :: a → c

(∘) :: (b → c) → (a → b) → (a → c)

g ∘ f = \x → g (f x)

- takes two functions that 'meet in the middle' and glues them together to form a third

Function composition

$$:: c \qquad :: b \rightarrow c \qquad :: b \qquad :: a \rightarrow b \qquad :: a$$

g (f x) ← [g] ← (f x) [f] ← x

g ∘ f :: a → c
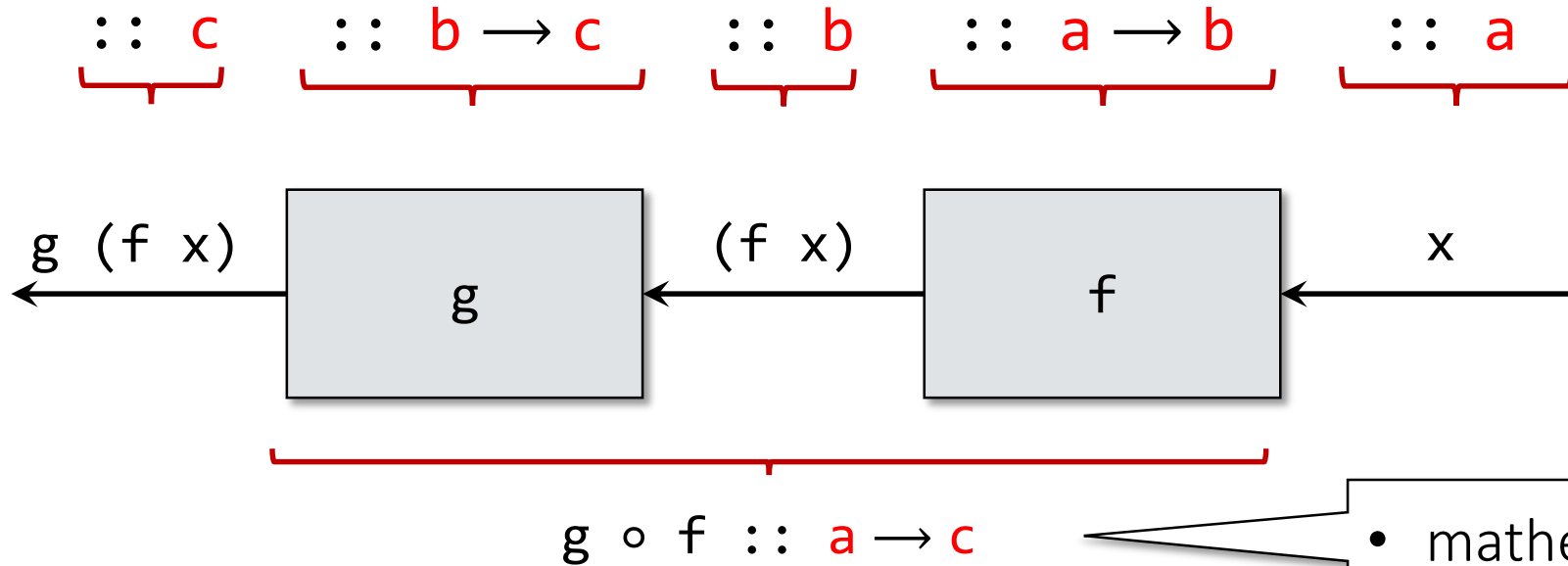
- mathematical convention
- pronounce: *g* <u>after</u> *f*

(∘) :: (b → c) → (a → b) → (a → c)

g ∘ f = \x → g (f x)

- takes two functions that 'meet in the middle' and glues them together to form a third

Folds

- many recursive definitions on lists share a pattern of computation
- capture that pattern as a function (abstraction, conciseness, general properties, familiarity, . . .)
- `map` and `filter` are two common patterns
- *folds* capture many more

A common pattern: foldr

```
sum :: Num p ⇒ [p] → p
sum []       = 0
sum (x : xs) = x + sum xs


product :: Num p ⇒ [p] → p
product []       = 1
product (x : xs) = x * product xs


and :: [Bool] → Bool
and []       = True
and (x : xs) = x && and xs


or :: [Bool] → Bool
or []       = False
or (x : xs) = x || or xs
```

A common pattern: foldr

```
sum :: Num p ⇒ [p] → p
sum []       = 0
sum (x : xs) = x + sum xs


product :: Num p ⇒ [p] → p
product []       = 1
product (x : xs) = x * product xs


and :: [Bool] → Bool
and []       = True
and (x : xs) = x && and xs


or :: [Bool] → Bool
or []       = False
or (x : xs) = x || or xs
```

A common pattern: foldr

```
sum :: Num p => [p] -> p
sum []       = 0
sum (x : xs) = x + sum xs
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr st ba []     = ba
foldr st ba (x:xs) = x `st` foldr st ba xs
```

```
product :: Num p => [p] -> p
product []       = 1
product (x : xs) = x * product xs
```

```
and :: [Bool] -> Bool
and []       = True
and (x : xs) = x && and xs
```

```
or :: [Bool] -> Bool
or []       = False
or (x : xs) = x || or xs
```

A common pattern: foldr

```
sum :: Num p => [p] -> p
sum []       = 0
sum (x : xs) = x + sum xs
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr st ba []     = ba
foldr st ba (x:xs) = x `st` foldr st ba xs
```

```
product :: Num p => [p] -> p
product []       = 1
product (x : xs) = x * product xs
```

```
and :: [Bool] -> Bool
and []       = True
and (x : xs) = x && and xs
```

```
sum     = foldr (+)  0
product = foldr (*)  1
and     = foldr (&&) True
or      = foldr (||) False
```

```
or :: [Bool] -> Bool
or []       = False
or (x : xs) = x || or xs
```

foldr — continued

```
foldr :: (a → b → b) → b → [a] → b
foldr st ba []     = ba
foldr st ba (x:xs) = x `st` foldr st ba xs
```

foldr — continued

```
foldr :: (a → b → b) → b → [a] → b
foldr st ba []      = ba
foldr st ba (x:xs) = x `st` foldr st ba xs
```

```
list   = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
```

foldr — continued

```
foldr :: (a → b → b) → b → [a] → b
foldr st ba []      = ba
foldr st ba (x:xs) = x `st` foldr st ba xs
```

```
         list  = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldr ⊕ e list  = 1 ⊕ ( 2 ⊕ ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []   = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise = filter p xs

[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []   = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise  = filter p xs

[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []   = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise = filter p xs

[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
       = foldr (const (+1)) 0
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []    = []
filter p (x:xs)
    | p x         = x : filter p xs
    | otherwise = filter p xs

[]        ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
       = foldr (const (+1)) 0

id = foldr (:) []
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []   = []
filter p (x:xs)
    | p x          = x : filter p xs
    | otherwise = filter p xs

[]         ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
       = foldr (const (+1)) 0

id = foldr (:) []

map f = foldr (\x ys → f x : ys) [ ]
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []   = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise = filter p xs

[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
       = foldr (const (+1)) 0

id = foldr (:) []

map f = foldr (\x ys → f x : ys) [ ]
      = foldr ((:) . f) []
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []   = []
filter p (x:xs)
   │  p x        = x : filter p xs
   │  otherwise = filter p xs

[]        ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
       = foldr (const (+1)) 0

id = foldr (:) []

map f = foldr (\x ys → f x : ys) [ ]
      = foldr ((:) . f) []

concat = foldr (++) []
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []   = []
filter p (x:xs)
    | p x        = x : filter p xs
    | otherwise = filter p xs

[]        ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
       = foldr (const (+1)) 0

id = foldr (:) []

map f = foldr (\x ys → f x : ys) [ ]
      = foldr ((:) . f) []

concat = foldr (++) []

reverse = foldr snoc [ ] where
            snoc x rs = rs ++ [x]
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []    = []
filter p (x:xs)
    | p x          = x : filter p xs
    | otherwise = filter p xs

[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
       = foldr (const (+1)) 0

id = foldr (:) []

map f = foldr (\x ys → f x : ys) [ ]
      = foldr ((:) . f) []

concat = foldr (++) []

reverse = foldr snoc [ ] where
             snoc x rs = rs ++ [x]
        = foldr (flip (++).(:[])) []
```

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []    = []
filter p (x:xs)
    | p x          = x : filter p xs
    | otherwise = filter p xs

[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
       = foldr (const (+1)) 0

id = foldr (:) []

map f = foldr (\x ys → f x : ys) [ ]
      = foldr ((:) . f) []

concat = foldr (++) []

reverse = foldr snoc [ ] where
          snoc x rs = rs ++ [x]
        = foldr (flip (++).(:[])) []

filter p = foldr (\x r → if p x then x:r else r) []
```

15

foldr examples

```
length []       = 0
length (x : xs) = 1 + length xs

id []       = []
id (x : xs) = x : id xs

map f []       = []
map f (x : xs) = f x : map f xs

concat []       = []
concat (x : xs) = x ++ concat xs

reverse []       = []
reverse (x : xs) = reverse xs ++ [x]

filter p []   = []
filter p (x:xs)
   | p x         = x : filter p xs
   | otherwise = filter p xs

[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length = foldr (\x y → y + 1) 0
       = foldr (const (+1)) 0

id = foldr (:) []

map f = foldr (\x ys → f x : ys) [ ]
      = foldr ((:) . f) []

concat = foldr (++) []

reverse = foldr snoc [ ] where
          snoc x rs = rs ++ [x]
        = foldr (flip (++).(:[])) []

filter p = foldr (\x r → if p x then x:r else r) []

xs ++ ys = foldr (:) ys xs
```

15

List design pattern and foldr

- task: define a function f :: $[P] \rightarrow S$
- step 1: solve the problem for the empty list

  f [] = ...

- step 2: solve the problem for non-empty lists;
  assume that you already have the solution for xs at hand; extend the intermediate solution to a solution for $x:xs$

  f [] = ...
  f (x:xs) = ... x ... xs ... f xs ...

List design pattern and foldr

- task: define a function f :: $[P] \longrightarrow S$

- step 1: solve the problem for the empty list

$$f \ [] \qquad = \ldots$$

- step 2: solve the problem for non-empty lists; assume that you already have the solution for `xs` at hand; extend the intermediate solution to a solution for `x:xs`

$$f \ [] \qquad = \ldots$$
$$f \ (x:xs) = \ldots \ x \ldots \ xs \ldots \ f \ xs \ldots$$

- suppose we don't need `xs` in step 2

List design pattern and foldr

- task: define a function f :: $[P] \rightarrow S$
- step 1: solve the problem for the empty list

$$f\ [] \quad = \ ...$$

- step 2: solve the problem for non-empty lists; assume that you already have the solution for `xs` at hand; extend the intermediate solution to a solution for `x:xs`

$$f\ [] \qquad = \ ...$$
$$f\ (x:xs) = \ ...\ x\ ...\ \cancel{xs}\ ...\ f\ xs\ ...$$

- suppose we don't need `xs` in step 2

List design pattern and foldr (2)

- task: define a function f `::` $[P]{\longrightarrow}S$
- step 1: solve the problem for the empty list

```
f []       = nil
```

- step 2: solve the problem for non-empty lists;
  assume that you already have the solution for `xs` at hand; extend the intermediate solution to a solution for `x:xs`

```
f []       = nil
f (x:xs) = step x (f xs)
```

List design pattern and foldr (2)

- task: define a function `f :: [P]`$\longrightarrow$`S`
- step 1: solve the problem for the empty list

  `f [] = nil`

- step 2: solve the problem for non-empty lists;
  assume that you already have the solution for `xs` at hand; extend the
  intermediate solution to a solution for `x:xs`

  `f [] = nil`
  `f (x:xs) = step x (f xs)`

- Then `f = foldr step nil`

Sorting: insertion sort

- given

```
insert :: (Ord a) ⟹ a → [a] → [a]
insert x [ ]   = [x]
insert x (y : ys)
    | x ≤ y      = x : y : ys
    | otherwise = y : insert x ys
```

- we have

```
insertionSort :: (Ord a) ⟹ [a] → [a]
insertionSort []      = []
insertionSort (x:xs) = insert x (insertionSort xs)
```

Sorting: insertion sort

- given

  insert :: (Ord a) $\Rightarrow$ a $\rightarrow$ [a] $\rightarrow$ [a]

  insert x [ ]   = [x]

  insert x (y : ys)

     | x $\leq$ y   = x : y : ys

     | **otherwise** = y : insert x ys

- we have

  insertionSort :: (Ord a) $\Rightarrow$ [a] $\rightarrow$ [a]

  insertionSort []   = []

  insertionSort (x:xs) = insert x (insertionSort xs)

- hence

  insertionSort = foldr insert [ ]

foldl

foldl

```
        list   = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldr ⊕ e list  = 1 ⊕ ( 2 ⊕ ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )
```

foldl

```
        list   = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldr ⊕ e list  = 1 ⊕ ( 2 ⊕ ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )
```

- not every list function is a **foldr**
  - e.g. `decimal [1,2,3] = 123`
- efficient algorithm using *Horner's rule*:
  - `decimal [1,2,3] = ((0 * 10 + 1) * 10 + 2) * 10 + 3`
- left-to-right computation— hence **foldl**

foldl

```
        list  = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldr ⊕ e list  = 1 ⊕ ( 2 ⊕ ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )
```

- not every list function is a **foldr**
  - e.g. `decimal [1,2,3] = 123`
- efficient algorithm using *Horner's rule*:
  - `decimal [1,2,3] = ((0 * 10 + 1) * 10 + 2) * 10 + 3`
- left-to-right computation— hence **foldl**

```
        list  = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
```

foldl

```
        list    = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldr ⊕ e list  = 1 ⊕ ( 2 ⊕ ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )
```

- not every list function is a **foldr**
  - e.g. `decimal [1,2,3] = 123`
- efficient algorithm using *Horner's rule*:
  - `decimal [1,2,3] = ((0 * 10 + 1) * 10 + 2) * 10 + 3`
- left-to-right computation— hence **foldl**

```
        list    = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldl ⊗ a list  = ( ( ( ( ( a ⊗ 1 ) ⊗ 2 ) ⊗ 3 ) ⊗ 4 ) ⊗ 5 )
```

foldl

```
        list  = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldr ⊕ e list  = 1 ⊕ ( 2 ⊕ ( 3 ⊕ ( 4 ⊕ ( 5 ⊕ e  ) ) ) ) )
```

- not every list function is a **foldr**
  - e.g. `decimal [1,2,3] = 123`
- efficient algorithm using *Horner's rule*:
  - `decimal [1,2,3] = ((0 * 10 + 1) * 10 + 2) * 10 + 3`
- left-to-right computation— hence **foldl**

```
        list  = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldl ⊗ a list  = ( ( ( ( ( a ⊗ 1 ) ⊗ 2 ) ⊗ 3 ) ⊗ 4 ) ⊗ 5 )
```

- definition of decimal
  ```
  decimal :: [Integer]→Integer
  decimal = foldl (\a n → a * 10 + n) 0
  ```

foldl — continued

```
foldl :: (b → a → b) → b → [a] → b
foldl op ac []     = ac
foldl op ac (x:xs) = foldl op (ac `op` x) xs
```

- example: **reverse**
  ```
  reverse :: [a]→[a]
  reverse  = foldr (\x xs→xs ++ [x]) [ ]
  ```

- another (more efficient) definition
  ```
  reverse' :: [a]→[a]
  reverse' = foldl (flip (:)) [ ]
  ```

reverse' = foldl (flip (:)) [ ]

```
                list = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldl ⊗          e  list = ( ( ( ( ( e ⊗ 1 ) ⊗ 2 ) ⊗ 3 ) ⊗ 4 ) ⊗ 5 )
foldl (flip (:)) [] list =
```

reverse' = foldl (flip (:)) [ ]

```
                list = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldl ⊗        e  list = ( ( ( ( ( e ⊗ 1 ) ⊗ 2 ) ⊗ 3 ) ⊗ 4 ) ⊗ 5 )
foldl (flip (:)) [] list =
 (((((([] (flip (:)) 1) (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
```

reverse' = foldl (flip (:)) [ ]

```
                 list = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldl ⊗        e  list = ( ( ( ( ( e ⊗ 1 ) ⊗ 2 ) ⊗ 3 ) ⊗ 4 ) ⊗ 5 )
foldl (flip (:)) [] list =
 (((((( [] (flip (:)) 1) (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
 (((((1:[])            (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
```

reverse' = foldl (flip (:)) [ ]

```
              list = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldl ⊗        e  list = ( ( ( ( ( e ⊗ 1 ) ⊗ 2 ) ⊗ 3 ) ⊗ 4 ) ⊗ 5 )
foldl (flip (:)) [] list =
 (((((([] (flip (:)) 1) (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
 (((((1:[])        (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
 ((((2:1:[])                     (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
```

reverse' = foldl (flip (:)) [ ]

```
                list = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldl ⊗        e  list = ( ( ( ( ( e ⊗ 1 ) ⊗ 2 ) ⊗ 3 ) ⊗ 4 ) ⊗ 5 )
foldl (flip (:)) [] list =
  (((((([] (flip (:)) 1) (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
  (((((1:[])          (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
  ((((2:1:[])                      (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
  (((3:2:1:[])                                  (flip (:)) 4) (flip (:)) 5) =
```

reverse' = foldl (flip (:)) [ ]

```
                list = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldl ⊗         e  list = ( ( ( ( ( e ⊗ 1 ) ⊗ 2 ) ⊗ 3 ) ⊗ 4 ) ⊗ 5 )
foldl (flip (:)) [] list =
  (((((([] (flip (:)) 1) (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
  (((((1:[])          (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
  ((((2:1:[])                     (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
  (((3:2:1:[])                                (flip (:)) 4) (flip (:)) 5) =
  ((4:3:2:1:[])                                           (flip (:)) 5) =
```

reverse' = foldl (flip (:)) [ ]

```
                    list = 1 : ( 2 : ( 3 : ( 4 : ( 5 : [] ) ) ) ) )
foldl ⊗          e  list = ( ( ( ( ( e ⊗ 1 ) ⊗ 2 ) ⊗ 3 ) ⊗ 4 ) ⊗ 5 )
foldl (flip (:)) [] list =
 (((((([] (flip (:)) 1) (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:)) 5) =
 ((((1:[])               (flip (:)) 2) (flip (:)) 3) (flip (:)) 4) (flip (:))  5) =
 ((((2:1:[])                            (flip (:)) 3) (flip (:)) 4) (flip (:))  5) =
 (((3:2:1:[])                                        (flip (:)) 4) (flip (:))  5) =
 ((4:3:2:1:[])                                                     (flip (:)) 5) =
 (5:4:3:2:1:[])
```

Producers

- so far we have focused on *consumers* (this seems to be close to the spirit of the time)

  - foldr consumes a list to provide a single value.

- *producers* are important too

- producers (unfolds) are *dual* to consumers (folds)

- **unfoldr** creates a list out of some seed.

Producers: some examples

Producers: some examples

```
-- repeat x is an infinite list, with x the value of every element.
repeat :: a → [a]
repeat x = x : repeat x
```

Producers: some examples

```
-- repeat x is an infinite list, with x the value of every element.
repeat :: a → [a]
repeat x = x : repeat x


-- iterate f x == [x, f x, f (f x), ...]
iterate :: (a → a) → a → [a]
iterate f x =  x : iterate f (f x)
```

Producers: some examples

```
-- repeat x is an infinite list, with x the value of every element.
repeat :: a → [a]
repeat x = x : repeat x


-- iterate f x == [x, f x, f (f x), ...]
iterate :: (a → a) → a → [a]
iterate f x =  x : iterate f (f x)


-- [n..] = [n,n+1,…] = enumFrom n
enumFrom :: Int → [Int]
enumFrom n = n : enumFrom (n+1)
```

Producers: some examples

```haskell
-- repeat x is an infinite list, with x the value of every element.
repeat :: a → [a]
repeat x = x : repeat x


-- iterate f x == [x, f x, f (f x), ...]
iterate :: (a → a) → a → [a]
iterate f x =  x : iterate f (f x)


-- [n..] = [n,n+1,…] = enumFrom n
enumFrom :: Int → [Int]
enumFrom n = n : enumFrom (n+1)


-- replicate n x is a list of length n with x the value of every element.
replicate :: Int → a → [a]
replicate 0 x =  []
replicate n x =  x : replicate (n-1) x
```

Producers: some examples

```haskell
-- repeat x is an infinite list, with x the value of every element.
repeat :: a → [a]
repeat x = x : repeat x


-- iterate f x == [x, f x, f (f x), ...]
iterate :: (a → a) → a → [a]
iterate f x =  x : iterate f (f x)


-- [n..] = [n,n+1,…] = enumFrom n
enumFrom :: Int → [Int]
enumFrom n = n : enumFrom (n+1)


-- replicate n x is a list of length n with x the value of every element.
replicate :: Int → a → [a]
replicate 0 x =  []
replicate n x =  x : replicate (n-1) x


-- [n..m] = [n,n+1,..,m] = enumFromTo n m
enumFromTo :: Int → Int → [Int]
enumFromTo n m

  | n <= m    = n : enumFromTo (n+1) m

  | otherwise = []
```

Common pattern: unfoldr

```
unfoldr :: (s → Maybe (a, s)) → s → [a]
unfoldr grow seed = case grow seed of
   Just (a, new_seed) → a : unfoldr grow new_seed
   Nothing            → []
```

unfoldr **examples**

```
repeat  x        = x : repeat x
repeatU x        = unfoldr (\s → Just (s,s)) x
```

unfoldr **examples**

```
repeat  x       = x : repeat x
repeatU x       = unfoldr (\s → Just (s,s)) x

iterate  f x    = x : iterate f (f x)
iterateU f x    = unfoldr (\s → Just (s, f s)) x
```

unfoldr **examples**

```
repeat  x       = x : repeat x
repeatU x       = unfoldr (\s ⟶ Just (s,s)) x

iterate  f x    = x : iterate f (f x)
iterateU f x    = unfoldr (\s ⟶ Just (s, f s)) x

enumFrom n      = n : enumFrom (n+1)
enumFromU n     = unfoldr (\s ⟶ Just (s,s+1)) n
```

unfoldr **examples**

```
repeat  x       = x : repeat x
repeatU x       = unfoldr (\s → Just (s,s)) x

iterate  f x    = x : iterate f (f x)
iterateU f x    = unfoldr (\s → Just (s, f s)) x

enumFrom n      = n : enumFrom (n+1)
enumFromU n     = unfoldr (\s → Just (s,s+1)) n

replicate 0 x   =  []
replicate n x   =  x : replicate (n-1) x
replicateU n x  = unfoldr (\n → if n==0 then Nothing else Just (x, n-1)) n
```

unfoldr **examples**

```
repeat   x      = x : repeat x
repeatU  x      = unfoldr (\s → Just (s,s)) x

iterate  f x    = x : iterate f (f x)
iterateU f x    = unfoldr (\s → Just (s, f s)) x

enumFrom n      = n : enumFrom (n+1)
enumFromU n     = unfoldr (\s → Just (s,s+1)) n

replicate 0 x   =  []
replicate n x   =  x : replicate (n-1) x
replicateU n x  = unfoldr (\n → if n==0 then Nothing else Just (x, n-1)) n

enumFromTo n m
  | n <= m       = n : enumFromTo (n+1) m
  | otherwise    = []

enumFromToU n m = unfoldr (\n → if n<=m then Just (n, n+1) else Nothing) n
```

unfoldr **examples**

```
repeat  x       = x : repeat x
repeatU x       = unfoldr (\s → Just (s,s)) x

iterate  f x    = x : iterate f (f x)
iterateU f x    = unfoldr (\s → Just (s, f s)) x

enumFrom n      = n : enumFrom (n+1)
enumFromU n     = unfoldr (\s → Just (s,s+1)) n

replicate 0 x   =  []
replicate n x   =  x : replicate (n-1) x
replicateU n x  = unfoldr (\n → if n==0 then Nothing else Just (x, n-1)) n

enumFromTo n m
  | n <= m      = n : enumFromTo (n+1) m
  | otherwise   = []

enumFromToU n m = unfoldr (\n → if n<=m then Just (n, n+1) else Nothing) n

mapU :: (a → b) → [a] → [b]

mapU f = unfoldr (\l → case l of [] → Nothing; x:xs → Just (f x, xs))
```

Sorting: selection sort

- select: put the smallest element in front
  ```
  select :: Ord a ⟹ [a] → [a]
  select []     = []
  select (x:xs) = swap x (select xs)
     where
         swap x [] = [x]
         swap x (y : ys)
           | x < y     = x:y:ys
           | otherwise = y:x:ys
  ```
- then we have
  ```
  selectionSort :: Ord a ⟹ [a] → [a]
  selectionSort l = case select l of
                      []     -> []
                      (x:xs) -> x : selectionSort xs
  ```

select **as a** foldr

```
select :: Ord a ⟹ [a] → (a,[a])
select []      = []
select (x:xs) = swap x (select xs)
   where
       swap x [] = [x]
       swap x (y : ys)
          | x < y      = x:y:ys
          | otherwise = y:x:ys
```

- **select** using **foldr**

```
selectF = foldr swap []
where
       swap x [] = [x]
       swap x (y : ys)
          | x < y      = x:y:ys
          | otherwise = y:x:ys
```

select **as a** foldr

```
select :: Ord a ⟹ [a] → (a,[a])
select []     = []
select (x:xs) = swap x (select xs)
  where
      swap x [] = [x]
      swap x (y : ys)
        | x < y      = x:y:ys
        | otherwise = y:x:ys
```

- select using **foldr**

```
selectF :: Ord a ⟹ [a] → [a]
selectF = foldr swap []
where
      swap x [] = [x]
      swap x (y : ys)
        | x < y      = x:y:ys
        | otherwise = y:x:ys
```

selectionSort **as an** unfoldr

```
selectionSort :: Ord a ⟹ [a] ⟶ [a]
selectionSort l = case select l of
                  []     -> []
                  (x:xs) -> x : selectionSort xs
```

selectionSort **as an** unfoldr

```
selectionSort :: Ord a ⟹ [a] → [a]
selectionSort l = case select l of
                      []     -> []
                      (x:xs) -> x : selectionSort xs
```

- **selectionSort** using **unfoldr**

```
selectionSortU :: Ord a ⟹ [a] → [a]
selectionSortU = unfoldr (uncons . select)
```

- **uncons** is defined in **Data.List**

selectionSort **as an** unfoldr

```
selectionSort :: Ord a ⟹ [a] → [a]
selectionSort l = case select l of
                    []     -> []
                    (x:xs) -> x : selectionSort xs
```

- **selectionSort** using **unfoldr**

```
selectionSortU :: Ord a ⟹ [a] → [a]
selectionSortU = unfoldr (uncons . select)
```

- **uncons** is defined in `Data.List`

```
uncons :: [a] -> Maybe (a, [a])
```

$\mathcal{O}(1)$. Decompose a list into its head and tail.

- If the list is empty, returns `Nothing`.
- If the list is non-empty, returns `Just (x, xs)`, where x is the head of the list and xs its tail.

```
>>> uncons []
Nothing
>>> uncons [1]
Just (1,[])
>>> uncons [1, 2, 3]
Just (1,[2,3])
```

putting it all together

```
selectionSort :: Ord a ⟹ [a] ⟶ [a]
selectionSort = unfoldr (uncons . foldr swap []) where
   swap x [] = [x]
   swap x (y : ys)
        | x < y      = x:y:ys
        | otherwise = y:x:ys
```

folding and unfolding trees

- externally-labelled binary trees (*leaf trees*)
  ```
  data LTree a = Tip a | Bin (LTree a) (LTree a)
  ```

folding and unfolding trees

- externally-labelled binary trees (*leaf trees*)
  `data LTree a = Tip a | Bin (LTree a) (LTree a)`

- LTree design pattern: define a function `f :: LTree P ⟶ S`

  - step 1: solve the problem for a leaf
    ```
    f (Tip n)          = … n …
    ```

  - step 2: solve the problem for internal nodes; assume that you already have the solutions for `l`, `r` at hand; extend the intermediate solution to a solution for `Bin l r`
    ```
    f (Tip n)   = … n …
    f (Bin l r) = … l … r … f l … f r …
    ```

folding and unfolding trees

- externally-labelled binary trees (*leaf trees*)
  data LTree a = Tip a | Bin (LTree a) (LTree a)

- LTree design pattern: define a function f :: LTree P ⟶ S

  - step 1: solve the problem for a leaf

    ```
    f (Tip n)           = … n …
    ```

  - step 2: solve the problem for internal nodes; assume that you already have the solutions for l, r at hand; extend the intermediate solution to a solution for Bin l r

    ```
    f (Tip n)   = … n …
    f (Bin l r) = … l … r … f l … f r …
    ```

- suppose you don't need l,r in step 2. Hence

folding and unfolding trees

- externally-labelled binary trees (*leaf trees*)
  data LTree a = Tip a | Bin (LTree a) (LTree a)

- LTree design pattern: define a function f :: LTree P ⟶ S

  - step 1: solve the problem for a leaf
    ```
    f (Tip n)          = … n …
    ```

  - step 2: solve the problem for internal nodes; assume that you already have the
    solutions for l, r at hand; extend the intermediate solution to a solution for Bin l r
    ```
    f (Tip n)   = … n …
    f (Bin l r) = … ⨉ … ⨉ … f l … f r …
    ```

- suppose you don't need l,r in step 2. Hence

folding and unfolding trees

- externally-labelled binary trees (*leaf trees*)
  `data LTree a = Tip a | Bin (LTree a) (LTree a)`
- LTree design pattern: define a function `f :: LTree P → S`
  - step 1: solve the problem for a leaf
    ```
    f (Tip n)            = … n …
    ```
  - step 2: solve the problem for internal nodes; assume that you already have the solutions for `l`, `r` at hand; extend the intermediate solution to a solution for `Bin l r`
    ```
    f (Tip n)    = … n …
    f (Bin l r) = … ✗ … ✗ … f l … f r …
    ```
- suppose you don't need `l,r` in step 2. Hence
    ```
    f (Tip n)    = tip n
    f (Bin l r) = bin (f l) (f r)
    ```

folding and unfolding trees

- externally-labelled binary trees (*leaf trees*)
  data `LTree a = Tip a | Bin (LTree a) (LTree a)`
- `LTree` design pattern: define a function `f :: LTree P ⟶ S`
  - step 1: solve the problem for a leaf
    ```
    f (Tip n)            = … n …
    ```
  - step 2: solve the problem for internal nodes; assume that you already have the solutions for `l`, `r` at hand; extend the intermediate solution to a solution for `Bin l r`
    ```
    f (Tip n)   = … n …
    f (Bin l r) = … ✗ … ✗ … f l … f r …
    ```
- suppose you don't need `l,r` in step 2. Hence
  ```
  f (Tip n)   = tip n
  f (Bin l r) = bin (f l) (f r)
  ```
- Then `f t = foldLTree bin tip t`

foldLTree

- definition

```
foldLTree :: (b → b → b) → (a → b) → LTree a → b
foldLTree bin tip = consume where
    consume (Tip e)   = tip e
    consume (Bin l r) = bin (consume l) (consume r)
```

foldLTree

- definition
```
foldLTree :: (b → b → b) → (a → b) → LTree a → b
foldLTree bin tip = consume where
    consume (Tip e)   = tip e
    consume (Bin l r) = bin (consume l) (consume r)
```
- examples
```
size :: (Num a) ⟹ LTree a -> a
size  = foldLTree (+) (\_ → 1)

sum :: (Num a) ⟹ LTree a → a
sum   = foldLTree (+) id

depth :: (Ord a, Num a) ⟹ LTree a → a
depth = foldLTree (\l r → max l r + 1) (\_ → 1)
```

Either

- the **Either** type

  data Either a b = Left a | Right b

- like **Maybe** often used for modeling failure

  ```
  head :: [a] ⟶ Either String a
  head []    = Left "head: empty list"
  head (x:_) = Right x
  ```

unfoldLTree

- unfolding a leaf tree

```
unfoldLTree :: (s → Either a (s, s)) → s → LTree a
unfoldLTree grow seed = produce seed where
  produce seed = case grow seed of
                    Left e              → Tip e
                    Right (lseed, rseed) → Bin (produce lseed) (produce rseed)
```

unfoldLTree

- unfolding a leaf tree

```
unfoldLTree :: (s → Either a (s, s)) → s → LTree a
unfoldLTree grow seed = produce seed where
    produce seed = case grow seed of
                    Left e              → Tip e
                    Right (lseed, rseed) → Bin (produce lseed) (produce rseed)
```

- a map for LTrees

```
mapLTree :: (a → b) → LTree a → LTree b
mapLTree f lt = unfoldLTree go lt where
-- go :: LTree a -> Either b (LTree a, LTree a)
```

unfoldLTree

- unfolding a leaf tree

```
unfoldLTree :: (s → Either a (s, s)) → s → LTree a
unfoldLTree grow seed = produce seed where
    produce seed = case grow seed of
                    Left e                → Tip e
                    Right (lseed, rseed) → Bin (produce lseed) (produce rseed)
```

- a map for LTrees

```
mapLTree :: (a → b) → LTree a → LTree b
mapLTree f lt = unfoldLTree go lt where
-- go :: LTree a -> Either b (LTree a, LTree a)
    go (Tip e)   = Left  (f e)
    go (Bin l r) = Right (l,r)
```

unfoldLTree

- unfolding a leaf tree

```
unfoldLTree :: (s → Either a (s, s)) → s → LTree a
unfoldLTree grow seed = produce seed where
    produce seed = case grow seed of
                    Left e                → Tip e
                    Right (lseed, rseed) → Bin (produce lseed) (produce rseed)
```

- a map for LTrees

```
mapLTree :: (a → b) → LTree a → LTree b
mapLTree f lt = unfoldLTree go lt where
-- go :: LTree a -> Either b (LTree a, LTree a)

    go (Tip e)   = Left  (f e)
    go (Bin l r) = Right (l,r)
```

- growing a balanced tree from a list

```
list2Tree :: [a] → LTree a
list2Tree = unfoldLTree (\s → case s of
                            [e]
                            xs
```

unfoldLTree

- unfolding a leaf tree

```
unfoldLTree :: (s → Either a (s, s)) → s → LTree a
unfoldLTree grow seed = produce seed where
    produce seed = case grow seed of
                      Left e                → Tip e
                      Right (lseed, rseed) → Bin (produce lseed) (produce rseed)
```

- a map for LTrees

```
mapLTree :: (a → b) → LTree a → LTree b
mapLTree f lt = unfoldLTree go lt where
-- go :: LTree a -> Either b (LTree a, LTree a)

    go (Tip e)   = Left  (f e)

    go (Bin l r) = Right (l,r)
```

- growing a balanced tree from a list

```
list2Tree :: [a] → LTree a
list2Tree = unfoldLTree (\s → case s of
                               [e] → Left e
                               xs  → Right (splitAt (length xs `div` 2) xs))
```

The art of functional programming

- question: what are the three most important concepts in programming?
- answer: abstraction, abstraction, abstraction!
- higher-order functions (HOFs) allow you to capture control structures, in particular, common patterns of recursion