Functional programming
Lecture 02 — Functions

Stéphane Vialette
stephane.vialette@univ-eiffel.fr

May 14, 2023

Laboratoire d'Informatique Gaspard-Monge, UMR CNRS 8049,
Université Gustave Eiffel

Lists

The anatomy of a list

- Lists are the workhorses of functional programming.
- Lists are inherently recursive.
- A list is either empty or an element followed by another list.

List notation

- The type `[a]` denotes lists of elements of type `a`.
- The empty list is denoted by `[]`.
- We can have lists over any type but we cannot mix different types in the same list

List notation

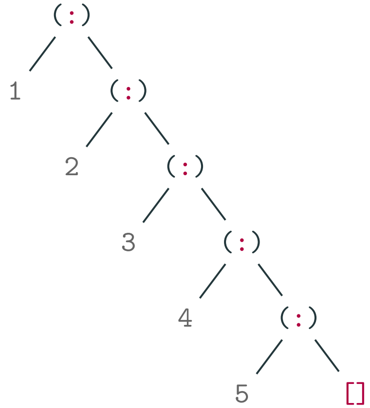
```
[]                              :: [a]

[undefined,undefined]           :: [a]

[sin,cos,tan]                   :: Floating a => [a -> a]

[[1,2,3],[4,5]]                 :: Num a => [[a]]

[(+1),(*2)]                     :: Num a => [a -> a]

[(1,'1',"1"),(2,'2',"2")]       :: Num a => [(a, Char, String)]

["tea","for",2]                 not valid
```

List notation

- The operator `(:) :: a -> [a] -> [a]` (pronounced cons) is constructor for lists.
- Cons associates to the right.
- Cons is non-strict in both arguments.
- List notation, such as `[1,2,3,4]`, is in fact an abbreviation for the more basic form `1:2:3:4:[]`

List notation

`[1,2,3,4,5]` $\equiv$ `1:2:3:4:5:[]`



First element

```
Data.List.head :: [a] -> a
```

`head` extracts the first element of a non-empty list.

First element

```
Data.List.head :: [a] -> a
```

`head` extracts the first element of a non-empty list.

```
λ> head [1,2,3,4]
1

λ> head (1:[2,3,4])
1

λ> head [1]
1

λ> head (1:[])
1

λ> head []
*** Exception: Prelude.head: empty list
```

First element

```
Data.List.head :: [a] -> a
```

`head` extracts the first element of a non-empty list.

```
head1 :: [a] -> []
head1 []     = error "*** Exception: head: empty list"
head1 (x:xs) = x

head2 :: [a] -> []
head2 []    = error "*** Exception: head: empty list"
head2 (x:_) = x
```

Except the first element

```
Data.List.tail :: [a] -> [a]
```

`tail` extracts the elements after the head of a non-empty list.

Except the first element

```
Data.List.tail :: [a] -> [a]
```

`tail` extracts the elements after the head of a non-empty list.

```
λ> tail [1,2,3,4]
[2,3,4]

λ> tail (1:[2,3,4])
[2,3,4]

λ> tail [1]
[]

λ> tail (1:[])
[]

λ> tail []
*** Exception: Prelude.tail: empty list
```

Except the first element

```
Data.List.tail :: [a] -> [a]
```

`tail` extracts the elements after the head of a non-empty list.

```
tail1 :: [a] -> []
tail1 []     = error "*** Exception: tail: empty list"
tail1 (x:xs) = xs

tail2 :: [a] -> []
tail2 []     = error "*** Exception: tail: empty list"
tail2 (_:xs) = xs
```

Enumerations

Enumerating lists of integers

When `m`, `n` and `p` are integers, we can write

`[m..n]`	for the list `[m,m+1,m+2,...,n]`
`[m..]`	for the infinite list `[m,m+1,m+2,...]`
`[m,p..n]`	for the list `[m,m+(p-n),m+2(p-n),...,n]`
`[m,p..]`	for the infinite list `[m,m+(p-n),m+2(p-n),...]`

Enumerating lists of integers

```
λ> [1..10]
[1,2,3,4,5,6,7,8,9,10]

λ> [10..1]
[]

λ> [1..]
[1,2,3,4,5,6,7,8,9,... ^CInterrupted.

λ> [1,3..9]
[1,3,5,7,9]

λ> [1,3..0]
[]
```

Enumerating lists of integers

```
λ> [10,8..0]
[10,8,6,4,2,0]

λ> [10,8..1]
[10,8,6,4,2]

λ> [5,3..]
[5,3,1,-1,-3,-5,-7,-9,... ^CInterrupted.
```

Enumerating lists of integers

Do not use floating point numbers in enumerations! Never ever!

```
λ> [0.1,0.3..1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]

λ> [1,0.6..0]
[1.0,0.6,0.19999999999999996]

λ> [1,4/3..2]
[1.0,1.3333333333333333,1.6666666666666665,1.9999999999999998]

λ> [5,13/3..3]
[5.0,4.333333333333333,3.666666666666666,2.999999999999999]
```

Enumerating lists of integers

Do not expect too much!

```
λ > [1,2,4,8,16..100] -- expecting the powers of 2 !
<interactive>: error: parse error on input '..'

λ > [2,3,5,7,11..101] -- expecting prime numbers
<interactive>: error: parse error on input '..'

λ > [1,-2,3,-4..9]    -- expecting [1,-2,3,-4,5,-6,7,-8,9]
<interactive>: error: parse error on input '..'

λ > [100,50,25..1]    -- expecting [100,50,25,12.5,6.25,...]
<interactive>: error: parse error on input '..'
```

Enumerating lists of integers

As a matter of fact, enumerations are not restricted to integers, but to members of yet another type class `Enum`.

Enumerating lists of integers

As a matter of fact, enumerations are not restricted to integers, but to members of yet another type class `Enum`.

`Char` is an instance of `Enum`:

```
λ> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

```
λ> succ 'a'
'b'
```

```
λ> pred 'z'
'y'
```

Enumerating lists of integers

As a matter of fact, enumerations are not restricted to integers, but to members of yet another type class `Enum`.

`Char` is an instance of `Enum`:

```
λ> ['A'..'Z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
λ> succ 'A'
'B'
```

```
λ> pred 'Z'
'Y'
```

Enumerating lists of integers

As a matter of fact, enumerations are not restricted to integers, but to members of yet another type class `Enum`.

`Char` is an instance of `Enum`:

```
λ> ['a','c'..'z']
"acegikmoqsuwy"

λ> ['z','y'..'a']
"zyxwvutsrqponmlkjihgfedcba"

λ> ['z','x'..'a']
"zxvtrpnljhfdb"
```

Enumerating lists of integers

As a matter of fact, enumerations are not restricted to integers, but to members of yet another type class `Enum`.

`Char` is an instance of `Enum`:

```
λ> succ 'Z'
'['

λ> pred 'a'
'`'

λ> ['A'..'z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz"
```

Enumerating lists of integers

As a matter of fact, enumerations are not restricted to integers, but to members of yet another type class `Enum`.

More on this soon !

List comprehensions

List comprehensions

Comprehensions are annotations in Haskell which are used to produce new lists from existing ones

```
[f x | x <- xs]
```

- Everything before the pipe determines the output of the list comprehension. It's basically what we want to do with the list elements.
- Everything after the pipe `|` is the generator.
- A generator:
  - Generates the set of values we can work with.
  - Binds each element from that set of values to `x` .
  - Draw our elements from that set (`<-` is pronounced "drawn from").

List comprehensions

- Set (*i.e.*, math) point of view.

  $\{x^2 \colon x \in \mathbb{N}\}$

- Comprehensions (*i.e.*, Haskell) point of view.

  ```
  [x*x | x <- [1..]]
  ```

List comprehensions

```
λ> [x*x | x <- [1..9]]
[1,4,9,16,25,36,49,64,81]

λ> [x*x | x <- [1,3..9]]
[1,9,25,49,81]

λ> [2^n | n <- [1..10]]
[2,4,8,16,32,64,128,256,512,1024]

λ> [(-1)^(n+1) * n | n <- [1..10]]
[1,-2,3,-4,5,-6,7,-8,9,-10]

λ> [100/n | n <- [1..10]]
[100.0,50.0,33.333333333333336,25.0,20.0,16.666666666666668,
 14.285714285714286,12.5,11.111111111111111,10.0]
```

Many generators

```
λ> [x | x <- []]
[]

λ> [(x,y) | x <- [1..3], y <- [1..3]]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)

λ> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]

λ> [x*y | x <- [1..3], y <- [1..3]]
[1,2,3,2,4,6,3,6,9]

λ> let n = 2 in [x*y `mod` n | x <- [1..3], y <- [1..3]]
[1,0,1,0,0,0,1,0,1]
```

Many lists

```
λ> [[1..n] | n <- [1..4]]
[[1],[1,2],[1,2,3],[1,2,3,4]]

λ> [[m..n] | m <- [1..4], n <- [1..4]]
[[1],[1,2],[1,2,3],[1,2,3,4],[],[2],[2,3],[2,3,4],[],[],[3],
 [3,4],[],[],[],[4]]

λ> [[m..n] | m <- [1..4], n <- [m..4]]
[[1],[1,2],[1,2,3],[1,2,3,4],[2],[2,3],[2,3,4],[3],[3,4],[4]]

λ> [[[m..n] | n <- [m..3]] | m <- [1..3]]
[[[1],[1,2],[1,2,3]],[[2],[2,3]],[[3]]]

λ> [[[m..n] | n <- [1..3]] | m <- [1..3]]
[[[1],[1,2],[1,2,3]],[[],[2],[2,3]],[[],[],[3]]]
```

Predicates

- If we do not want to draw all elements from a list, we can add a condition, a predicate.
- A predicate is a function which takes an element and returns a boolean value.

```
[f x | x <- xs, p1 x, p2 x, ..., pn x]
```

Predicates

```
λ> [x*x | x <- [1..10], even x]
[4,16,36,64,100]

λ> [(x,x*x) | x <- [1..10], even x]
[(2,4),(4,16),(6,36),(8,64),(10,100)]

λ> [(x,x*x) | x <- [1..10], even x, x `mod` 3 /= 0]
[(2,4),(4,16),(8,64),(10,100)]

λ>  [(x, y) | x <- [1..10], even x, y <- [x..10], odd y]
[(2,3),(2,5),(2,7),(2,9),(4,5),(4,7),(4,9),(6,7),(6,9),(8,9)]

λ> [x | x <- [1..100], even x, x `mod` 3 == 0, x `mod` 5 == 0]
[30,60,90]
```

Predicates and pattern matching

```
λ > [x | (x,1) <- [(x,y) | x <- [1..3], y <- [1..3]]]
[1,2,3]

λ > [x | (x,y) <- [(x,y) | x <- [1..3], y <- [1..3]], y<=2]
[1,1,2,2,3,3]

λ > [(x,y) | (x,y) <- [(x,y) | x <- [1..3], y <- [1..3]], x==y]
[(1,1),(2,2),(3,3)]

λ > [y | xys <- [[(x,x*2)] | x <- [1..6]], (2,y) <- xys]
[4]

λ > [y | xys <- [[(x,x*2)] | x <- [1..6]], (x,y) <- xys, even x]
[4,8,12]
```

Problem solving with list comprehensions

Compute the list `[1,1+2,...,1+2+3+...+n]`.

```haskell
-- assuming we don't know about Data.Foldable.sum
sums :: (Num a, Enum a, Eq a) => a -> [a]
sums n = [f k | k <- [1..n]]
  where
    f 1 = 1
    f k = k + f (k-1)
```

```
λ> sums 10
[1,3,6,10,15,21,28,36,45,55]
```

```
λ> [n*(n+1) `div` 2  | n <- [1..10]]
[1,3,6,10,15,21,28,36,45,55]
```

Problem solving with list comprehensions

Compute the list `[1^2,1^2+2^2,...,1^2+2^2+3^2+...+n^2]`.

```
-- assuming we don't know about Data.Foldable.sum
sumsSq :: (Num a, Enum a, Eq a) => a -> [a]
sumsSq n = [f k | k <- [1..n]]
  where
    f 1 = 1
    f k = k*k + f (k-1)
```

```
λ> sumsSq 10
[1,5,14,30,55,91,140,204,285,385]
```

```
λ> [n*(n+1)*(2*n+1) `div` 6 | n <- [1..10]]
[1,5,14,30,55,91,140,204,285,385]
```

Problem solving with list comprehensions

Compute the list of all positive intergers $k \leqslant n$ such that $k \not\equiv 0 \pmod 2$, $k \not\equiv 0 \pmod 3$, $k \equiv 1 \pmod 5$ and $k \equiv 0 \pmod 7$.

```
f :: Integral a => a -> [a]
f n = [k | k <- [1..n]
         , odd k
         , k `mod` 3 > 0
         , k `mod` 5 == 1
         , k `mod` 7 == 0]
```

```
λ> f 1000
[91,161,301,371,511,581,721,791,931]
```

Problem solving with list comprehensions

A Pythagorean triple consists of three positive integers a, b, and c, such that $a^2 + b^2 = c^2$. Compute all Pythagorean triples with $a < b < c \leqslant 15$.

```
-- naive implementation
pythT :: (Num a, Enum a, Eq a) => c -> [(a, a, a)]
pythT n = [(a, b, c) | a <- [1..n]
                     , b <- [a+1..n]
                     , c <- [b+1..n]
                     , a*a + b*b == c*c]
```

```
λ> pythT 15
[(3,4,5),(5,12,13),(6,8,10),(9,12,15)]
```

Problem solving with list comprehensions

Compute the infinite list of the powers of 2.

```haskell
p2s1 :: Num a => [a]
p2s1 = [2^n | n <- 1:p2s1]
```

```
λ> take 11 p2s1
[2,4,8,16,32,64,128,256,512,1024,2048]

λ> head (drop 120 p2s1)
2658455991569831745807614120560689152
```

Problem solving with list comprehensions

Compute the infinite list of the powers of 2.

```
p2s2 :: Num a => [a]
p2s2 = [2*n | n <- 1:p2s2]

n=1
     p2s2 = 1:2^1:p2s2
n=2
     p2s2 = 1:2^1:2^2:p2s2
n=3
     p2s2 = 1:2^1:2^2:2^3:p2s2
n=4
     p2s2 = 1:2^1:2^2:2^3:2^4:p2s2
...
```

Problem solving with list comprehensions

Compute the infinite list of all binary strings.

```
binaries :: [String]
binaries = [b:bs | bs <- "":binaries, b <- ['0','1']]
```

```
λ> take 11 binaries
["0","1","00","10","01","11","000","100","010","110","001"]

λ> head (drop 10000000 binaries)
"01000001011010010001100"
```

Processing lists – basic functions

Finding

```
Data.List.elem :: (Eq a) => a -> [a] -> Bool
```

`elem` is the list membership predicate, usually written in infix form, e.g., `x `elem` xs`. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

Finding

```
Data.List.elem :: (Eq a) => a -> [a] -> Bool
```

`elem` is the list membership predicate, usually written in infix form, e.g., `` x `elem` xs ``. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

```
λ> 2 `elem` [1..5]
True
```

```
λ> 8 `elem` [1..5]
False
```

Finding

```
Data.List.elem :: (Eq a) => a -> [a] -> Bool
```

`elem` is the list membership predicate, usually written in infix form, e.g., `x `elem` xs`. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

```
elem1 :: Eq a => a -> [a] -> Bool
elem1 _  []     = False
elem1 x' (x:xs)
  | x == x'    = True
  | otherwise = elem1 x' xs

elem2 :: Eq a => a -> [a] -> Bool
elem2 _  []      = False
elem2 x' (x:xs) = x == x' || elem2 x' xs
```

Repeating

```
Data.List.repeat :: a -> [a]
```

`repeat` takes an element and returns an infinite list that just has that element.

Repeating

```haskell
Data.List.repeat :: a -> [a]
```

`repeat` takes an element and returns an infinite list that just has that element.

```
λ> repeat 'a'
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
^C Interrupted.
```

```
λ> repeat "a" -- i.e. repeat ['a']
["a","a","a","a","a","a","a","a"...
^C Interrupted.
```

Repeating

```haskell
Data.List.repeat :: a -> [a]
```

`repeat` takes an element and returns an infinite list that just has that element.

```haskell
repeat1 :: a -> [a]
repeat1 x = x:repeat1 x

repeat2 :: a -> [a]
repeat2 x = [x | n <- [1 ..]]

repeat3 :: a -> [a]
repeat3 x = [x | _ <- [1 ..]]
```

Taking

```
Data.List.take :: Int -> [a] -> [a]
```

`take` takes a certain number of elements from a list.

Taking

```
Data.List.take :: Int -> [a] -> [a]
```

`take` takes a certain number of elements from a list.

```
λ> take 10 [1..20]
[1,2,3,4,5,6,7,8,9,10]

λ> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]

λ> take 20 [1..10]
[1,2,3,4,5,6,7,8,9,10]

λ> take 0 [1..]
[]

λ> take (-1) [1..]
[]
```

Taking

```
Data.List.take :: Int -> [a] -> [a]
```

`take` takes a certain number of elements from a list.

```
take1 :: (Ord t, Num t) => t -> [a] -> [a]
take1 _ [] = []
take1 n (x:xs)
  | n <= 0     = []
  | otherwise = x:take1 (n-1) xs

take2 :: (Eq t, Num t) => t -> [a] -> [a]
take2 _ [] = []
take2 0 _  = []
take2 n (x:xs) = x:take2 (n-1) xs
```

Dropping

```
Data.List.drop :: Int -> [a] -> [a]
```

`drop` drops a certain number of elements from a list.

Dropping

```
Data.List.drop :: Int -> [a] -> [a]
```

`drop` drops a certain number of elements from a list.

```
λ> drop 10 [1..20]
[11,12,13,14,15,16,17,18,19,20]

λ> drop 10 [1..]
[11,12,13,14,15,16,17,18,19,20,...
^C Interrupted.

λ> drop (-1) [1..]
[1,2,3,4,5,6,7,8,9,10,...
^C Interrupted.

λ> drop 20 [1..10]
[]
```

Dropping

```
Data.List.drop :: Int -> [a] -> [a]
```

`drop` drops a certain number of elements from a list.

```
drop1 :: (Ord t, Num t) => t -> [a] -> [a]
drop1 _ [] = []
drop1 n (x:xs)
  | n > 0     = drop1 (n-1) xs
  | otherwise = x:xs

drop2 :: (Ord t, Num t) => t -> [a] -> [a]
drop2 _ [] = []
drop2 n xs@(_:xs')
  | n > 0     = drop2 (n-1) xs'
  | otherwise = xs
```

Taking and Dropping – In practice

Define a function that rotates the elements of a list `n` places to the left, wrapping around at the start of the list, and assuming that the integer argument `n` is between zero and the length of the list.

For example:

```
λ > rotate 0 [1..8]
[1,2,3,4,5,6,7,8]

λ > rotate 1 [1..8]
[2,3,4,5,6,7,8,1]

λ > rotate 4 [1..8]
[5,6,7,8,1,2,3,4]
```

Taking and Dropping – In practice

Define a function that rotates the elements of a list n places to the left, wrapping around at the start of the list, and assuming that the integer argument n is between zero and the length of the list.

```
rotate1 :: Int -> [a] -> [a]
rotate1 = go []
  where
    go acc 0 xs      = xs ++ reverse acc
    go acc n []      = go [] n (reverse acc) -- (*)
    go acc n (x:xs) = go (x:acc) (n-1) xs
```

```
rotate2 :: Int -> [a] -> [a]
rotate2 n xs = drop n xs ++ take n xs
```

Replicating

```
Data.List.replicate :: Int -> a -> [a]
```

`replicate` takes an `Int` and some element and returns a list that has several repetitions of the same element.

Replicating

```
Data.List.replicate :: Int -> a -> [a]
```

`replicate` takes an `Int` and some element and returns a list that has several repetitions of the same element.

```
λ> replicate 10 1
[1,1,1,1,1,1,1,1,1,1]

λ> replicate 0 1
[]

λ> replicate (-1) 1
[]
```

Replicating

```
Data.List.replicate :: Int -> a -> [a]
```

`replicate` takes an `Int` and some element and returns a list that has several repetitions of the same element.

```
replicate1 :: (Num t, Ord t) => t -> a -> [a]
replicate1 n x
    | n <= 0    = []
    | otherwise = x:replicate1 (n-1) x

replicate2 :: (Ord t, Num t) => t -> a -> [a]
replicate2 n x = take n (repeat x)
```

Suffixing

```
Data.List.tails :: [a] -> [[a]]
```

`tails` returns all final segments of the argument, longest first.

Suffixing

```
Data.List.tails :: [a] -> [[a]]
```

`tails` returns all final segments of the argument, longest first.

```
λ> tails [1..4]
[[1,2,3,4],[2,3,4],[3,4],[4],[]]

λ> tails []
[[]]

λ> tails [1..]
^C Interrupted.

λ> head (tails [1..])
^C Interrupted.
```

Suffixing

```
Data.List.tails :: [a] -> [[a]]
```

`tails` returns all final segments of the argument, longest first.

```
tails1 :: [a] -> [[a]]
tails1 [] = [[]]
tails1 (x:xs) = (x:xs):tails1 xs

tails2 :: [a] -> [[a]]
tails2 [] = [[]]
tails2 xs@(_:xs') = xs:tails2 xs'
```

Reversing

```
Data.List.reverse :: [a] -> [a]
```

`reverse xs` returns the elements of `xs` in reverse order. `xs` must be finite.

Reversing

```
Data.List.reverse :: [a] -> [a]
```

`reverse xs` returns the elements of `xs` in reverse order. `xs` must be finite.

```
λ> reverse [1..5]
[5,4,3,2,1]

λ> reverse []
[]

λ> reverse [1..]
^C Interrupted.
```

Reversing

```
Data.List.reverse :: [a] -> [a]
```

`reverse xs` returns the elements of `xs` in reverse order. `xs` must be finite.

```
-- inefficient because of (++)
reverse1 :: [a] -> [a]
reverse1 [] = []
reverse1 (x:xs) = reverse1 xs ++ [x]

-- using an accumulator is much more efficient
reverse2 :: [a] -> [a]
reverse2 = go []
  where
    go acc []     = acc
    go acc (x:xs) = go (x:acc) xs
```

Cutting last

```
Data.List.init :: [a] -> [a]
```

`init` returns all the elements of a list except the last one. The list must be non-empty.

Cutting last

```
Data.List.init :: [a] -> [a]
```

`init` returns all the elements of a list except the last one. The list must be non-empty.

```
λ> init [1,2,3,4]
[1,2,3]

λ> init [1]
[]

λ> init []
*** Exception: Prelude.init: empty list
```

Cutting last

```
Data.List.init :: [a] -> [a]
```

`init` returns all the elements of a list except the last one. The list must be non-empty.

```
init1 :: [a] -> [a]
init1 []     = error "*** Exception: init': empty list"
init1 [_]    = []
init1 (x:xs) = x:init' xs

-- with functors and Maybe type
safeInit :: [a] -> Maybe [a]
safeInit []     = Nothing
safeInit [_]    = Just []
safeInit (x:xs) = (x :) <$> safeInit xs
```

Prefixing

```
Data.List.inits :: [a] -> [[a]]
```

`inits` returns all initial segments of the argument, shortest first.

Prefixing

```
Data.List.inits :: [a] -> [[a]]
```

`inits` returns all initial segments of the argument, shortest first.

```
λ> inits [1..4]
[[],[1],[1,2],[1,2,3],[1,2,3,4]]

λ> inits [1]
[[],[1]]

λ> inits []
[[]]

λ> inits [1..]
[[],[1],[1,2],[1,2,3],[1,2,3,4],...^C Interrupted.

λ> head (inits [1..])
[]
```

Prefixing

```
Data.List.inits :: [a] -> [[a]]
```

`inits` returns all initial segments of the argument, shortest first.

```
inits1 :: [a] -> [[a]]
inits1 [] = [[]]
inits1 xs = inits1 (init xs) ++ [xs]

inits2 :: [a] -> [[a]]
inits2 = reverse . go
  where
    go [] = [[]]
    go xs = xs:go (init xs)
```

Interspersing

`Data.List.intersperse :: a -> [a] -> [a]`

`intersperse` takes an element and a list and intersperses that element between the elements of the list.

Interspersing

```
Data.List.intersperse :: a -> [a] -> [a]
```

`intersperse` takes an element and a list and intersperses that element between the elements of the list.

```
λ> intersperse ',' ['a','b','c','d']
"a,b,c,d"

λ> intersperse 0 [1,2,3,4]
[1,0,2,0,3,0,4]

λ> intersperse [0] [[1,2],[3,4],[5,6]]
[[1,2],[0],[3,4],[0],[5,6]]
```

Interspersing

```
Data.List.intersperse :: a -> [a] -> [a]
```

`intersperse` takes an element and a list and intersperses that element between the elements of the list.

```
intersperse1 :: a -> [a] -> [a]
intersperse1 _ []     = []
intersperse1 _ [x]    = [x]
intersperse1 y (x:xs) = x:y:intersperse1 i xs
```

Concatening

```
Data.List.concat :: Foldable t => t [a] -> [a]
```

`concat` concatenates a list of lists.

Concatening

```
Data.List.concat :: Foldable t => t [a] -> [a]
```

`concat` concatenates a list of lists.

```
λ> concat [[1,2],[3,4],[5,6]]
[1,2,3,4,5,6]

λ> concat [[1,2]]
[1,2]

λ> concat [[]]
[]

λ> concat []
[]
```

Concatening

```
Data.List.concat :: Foldable t => t [a] -> [a]
```

`concat` concatenates a list of lists.

```
-- recursive
concat1 :: [[a]] -> [a]
concat1 []       = []
concat1 (xs:xss) = xs ++ concat1 xss

-- with a list comprehension
concat2 :: [[a]] -> [a]
concat2 xss = [x | xs <- xss, x <- xs]
```

Intercalating

```
Data.List.intercalate :: [a] -> [[a]] -> [a]
```

`intercalate xs xss` inserts the list `xs` in between the lists in `xss` and concatenates the result.

Intercalating

```
Data.List.intercalate :: [a] -> [[a]] -> [a]
```

`intercalate xs xss` inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
λ> intercalate [0] [[1,2],[3,4],[5,6]]
[1,2,0,3,4,0,5,6]

λ> intercalate [0] [[1,2]]
[1,2]

λ> intercalate [0] []
[]

λ> intercalate " -> " ["task1","task2","task3"]
"task1 -> task2 -> task3"
```

Intercalating

```
Data.List.intercalate :: [a] -> [[a]] -> [a]
```

`intercalate xs xss` inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
intercalate1  :: [a] -> [[a]] -> [a]
intercalate1 _   []       = []
intercalate1 _   [xs]     = xs
intercalate1 xs' (xs:xss) = xs  ++
                            xs' ++
                            intercalate1 xs' xss

intercalate2 :: [a] -> [[a]] -> [a]
intercalate2 xs xss = concat (intersperse xs xss)
```

Zipping

```
Data.List.zip :: [a] -> [b] -> [(a, b)]
```

`zip` takes two lists and returns a list of corresponding pairs.

Zipping

```
Data.List.zip :: [a] -> [b] -> [(a, b)]
```

`zip` takes two lists and returns a list of corresponding pairs.

```
λ> zip [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]

λ> zip [1,2,3,4] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]

λ> zip [1,2,3] ['a','b','c','d']
[(1,'a'),(2,'b'),(3,'c')]
```

Zipping

```
Data.List.zip :: [a] -> [b] -> [(a, b)]
```

`zip` takes two lists and returns a list of corresponding pairs.

```
zip :: [a] -> [b] -> [(a, b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

Zipping – In practice

Index a list from a given integer.

```
λ > index 0 ['a'..'f']
[(0,'a'),(1,'b'),(2,'c'),(3,'d'),(4,'e'),(5,'f')]

λ > index 1 ['a'..'f']
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e'),(6,'f')]

λ > index (2^10) ['a'..'e']
[(1024,'a'),(1025,'b'),(1026,'c'),(1027,'d'),(1028,'e')]

λ > index2 (-10) ['a'..'f']
[(-10,'a'),(-9,'b'),(-8,'c'),(-7,'d'),(-6,'e'),(-5,'f')]
```

Zipping – In practice

Index a list from a given integer.

```
index1 :: Num a => a -> [b] -> [(a, b)]
index1 n []     = []
index1 n (x:xs) = (n,x):index1 (n+1) xs

index2 :: Enum a => a -> [b] -> [(a, b)]
index2 n xs = zip [n..] xs
```

Zipping – In practice

Implementing `take` with `zip`.

```
take3 :: (Num a, Enum a, Ord a) => a -> [b] -> [b]
take3 n xs = go (zip xs [1..])
  where
    go ((x,i):xis)
      | i <= n    = x:go xis
      | otherwise = []

take4 :: (Num a, Enum a, Ord a) => a -> [b] -> [b]
take4 n xs = go $ zip xs [1..]
  where
    go ((x,i):xis)
      | i <= n    = x:go xis
      | otherwise = []
```

Zipping – In practice

Implementing `take` with `zip`.

```
-- don't do this!!!
-- infinite computation: a predicate does not stop
-- the infinite enumeration (we are just skipping
-- values again and again).
take5 :: (Num a, Enum a, Ord a) => a -> [b] -> [b]
take5 n xs = [x | (x, i) <- zip xs [1..], i <= n]

-- not better!
take5 :: Int -> [a] -> [a]
take5 n xs = [x | (x, i) <- zip xs [1..nxs], i <= n]
  where
    nxs = length xs
```

Anding

```
Data.Foldable.and :: Foldable t => t Bool -> Bool
                                   [Bool] -> Bool
```

`and` returns the conjunction of a Boolean list, the result can be True only for finite lists

Anding

```
Data.Foldable.and :: Foldable t => t Bool -> Bool
                                   [Bool] -> Bool
```

`and` returns the conjunction of a Boolean list, the result can be True only for finite lists

```
λ> and []
True

λ> and [True]
True

λ> and [False]
False

λ> and (take 100 (repeat True) ++ [False])
False
```

Anding

```
Data.Foldable.and :: Foldable t => t Bool -> Bool
                                   [Bool] -> Bool
```

and returns the conjunction of a Boolean list, the result can be True only for finite lists

```
and1 :: [Bool] -> Bool
and1 []          = True
and1 (False:bas) = False
and1 (True:bs)   = and1 bs

and2 []     = True
and2 (b:bs) = b && and2 bs
```

Oring

```
Data.Foldable.or :: Foldable t => t Bool -> Bool
                                  [Bool] -> Bool
```

`or` returns the disjunction of a Boolean list, the result can be True only for finite lists

Oring

```
Data.Foldable.or :: Foldable t => t Bool -> Bool
                                  [Bool] -> Bool
```

`or` returns the disjunction of a Boolean list, the result can be True only for finite lists

```
λ> or []
False
```

```
λ> or [True]
True
```

```
λ> or (take 100 (repeat False))
False
```

```
λ> or (take 100 (repeat False) ++ [True])
True
```

Oring

```
Data.Foldable.or :: Foldable t => t Bool -> Bool
                                  [Bool] -> Bool
```

`or` returns the disjunction of a Boolean list, the result can be True only for finite lists

```
or1 :: [Bool] -> Bool
or1 []          = False
or1 (True:bas) = True
or1 (False:bs) = or1 bs
```

```
or2 :: [Bool] -> Bool
or2 []      = False
or2 (b:bs) = b || or2 bs
```

Maximizing

```
Data.Foldable.maximum :: (Foldable t, Ord a) => t a -> a
 [a] -> a
```

`maximum` returns the largest element of a non-empty structure. (`minimum` returns the largest element of a non-empty structure).

```
λ> maximum []
*** Exception: Prelude.maximum: empty list

λ> maximum [1]
1

λ> maximum [4,3,7,1,8,6,2,3,5]
8

λ> maximum [2,3,1,4,3,1,2,4]
4
```

Maximizing

```haskell
Data.Foldable.maximum :: (Foldable t, Ord a) => t a -> a
 [a] -> a

maximum1 :: Ord a => [a] -> a
maximum1 []     = error "empty list"
maximum1 [x]    = x
maximum1 (x:xs) = let m = maximum1 xs in if m>x then m else

maximum2 :: Ord a => [a] -> a
maximum2 []     = error "empty list"
maximum2 [x]    = x
maximum2 (x:xs) = max x (maximum2 xs)
```

Maximizing

```
Data.Foldable.maximum :: (Foldable t, Ord a) => t a -> a
 [a] -> a
```

```
maximum3 :: Ord a => [a] -> a
maximum3 []       = error "empty list"
maximum3 (x:xs) = go x xs
  where
    go m []         = m
    go m (x':xs')
      | x' > m      = go x' xs'
      | otherwise = go m xs'
```

High-order functions

High-order functions

- A function that takes a function as an argument or returns a function as a result is called a high-order function.
- Because the term curried already exists for returning functions as results, the ther high-order is often just used for taking functions as arguments.
- Using high-order functions considerably increases the power of Haskell by allowing common programming patterns to be encapsulated as functions within the language itself.

Filtering

```
Data.List.filter :: (a -> Bool) -> [a] -> [a]
```

`filter` applied to a predicate and a list, returns the list of those elements that satisfy the predicate.

Filtering

```
Data.List.filter :: (a -> Bool) -> [a] -> [a]
```

`filter` applied to a predicate and a list, returns the list of those elements that satisfy the predicate.

```
λ > filter even [1..10]
[2,4,6,8,10]
λ > filter (\x -> x `mod` 2 == 0) [1..10]
[2,4,6,8,10]
λ > filter (\x -> even x && odd x) [1..10]
[]
λ > filter (> 5) [1,5,2,6,3,7,4,8]
[6,7,8]
λ > filter (<= 5) [1,5,2,6,3,7,4,8]
[1,5,2,3,4]
```

Filtering

```
Data.List.filter :: (a -> Bool) -> [a] -> [a]
```

`filter` applied to a predicate and a list, returns the list of those elements that satisfy the predicate.

```
-- recursive
filter1 :: (a -> Bool) -> [a] -> [a]
filter1 _ [] = []
filter1 p (x:xs)
  | p x       = x:filter1 p xs
  | otherwise = filter1 p xs

-- with a list comprehension
filter2 :: (a -> Bool) -> [a] -> [a]
filter2 p xs = [x | x <- xs, p x]
```

Mapping

```
Data.List.map :: (a -> b) -> [a] -> [b]
```

`map f xs` is the list obtained by applying `f` to each element of `xs`.

Mapping

```
Data.List.map :: (a -> b) -> [a] -> [b]
```

`map f xs` is the list obtained by applying `f` to each element of `xs`.

```
λ> map (*2) [1..5]
[2,4,6,8,10]
λ> map even [1..5]
[False,True,False,True,False]
λ> map (\x -> 2*x) [1..5] -- equiv map (2*) [1..5]
[2,4,6,8,10]
λ> map (\x -> [x]) [1..5]
[[1],[2],[3],[4],[5]]
```

Mapping

```
Data.List.map :: (a -> b) -> [a] -> [b]
```

`map f xs` is the list obtained by applying `f` to each element of `xs`.

```
λ> map (map (* 2)) [[1,2,3],[4,5,6],[7,8,9]]
[[2,4,6],[8,10,12],[14,16,18]]
λ> map (filter even) [[1,2,3],[4,5,6],[7,8,9]]
[[2],[4,6],[8]]
λ> map length [[1,2,3],[4,5,6],[7,8,9]]
[3,3,3]
λ> map (take 2) [[1,2,3],[4,5,6],[7,8,9]]
[[1,2],[4,5],[7,8]]
```

Mapping

```
Data.List.map :: (a -> b) -> [a] -> [b]
```

`map f xs` is the list obtained by applying `f` to each element of `xs`.

```
-- recursive
map1 :: (a -> b) -> [a] -> [b]
map1 _ [] = []
map1 f (x:xs) = f x:map1 f xs

-- with a list comprehension
map2 :: (a -> b) -> [a] -> [b]
map1 f xs = [f x | x <- xs]
```

Mapping – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

Mapping – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \qquad M' = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
m = [[1,2],
     [3,4],
     [5,6]]
```

```
m' = [[1,2,0,0,0,0,0],
      [3,4,0,0,0,0,0],
      [5,6,0,0,0,0,0]]
```

Mapping – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

```
λ> m = [[1,2],[3,4],[5,6]]
λ> addExtraColumns 0 m
[[1,2],[3,4],[5,6]]
λ> addExtraColumns 1 m
[[1,2,0],[3,4,0],[5,6,0]]
λ> addExtraColumns 5 m
[[1,2,0,0,0,0,0],[3,4,0,0,0,0,0],[5,6,0,0,0,0,0]]
```

Mapping – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

```
addExtraColumns1 :: Num a => Int -> [[a]] -> [[a]]
addExtraColumns1 k xss = map (++ yss) xss
  where
    yss = replicate k 0
```

Taking with a predicate

```
Data.List.takeWhile :: (a -> Bool) -> [a] -> [a]
```

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`.

Taking with a predicate

```
Data.List.takeWhile :: (a -> Bool) -> [a] -> [a]
```

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`.

```
λ> takeWhile (< 10) [1..20]
[1,2,3,4,5,6,7,8,9]
λ> takeWhile odd ([1,3..10] ++ [1..10])
[1,3,5,7,9,1]
λ> takeWhile even [1..10]
[]
λ> takeWhile (> 0) (map (`mod` 5) [1..10])
[1,2,3,4]
```

Taking with a predicate

```haskell
Data.List.takeWhile :: (a -> Bool) -> [a] -> [a]
```

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`.

```haskell
takeWhile1 :: (a -> Bool) -> [a] -> [a]
takeWhile1 _ [] = []
takeWhile1 p (x:xs)
  | p x         = x:takeWhile1 p xs
  | otherwise   = []
```

Dropping with a predicate

```
Data.List.dropWhile :: (a -> Bool) -> [a] -> [a]
```

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`.

Dropping with a predicate

```haskell
Data.List.dropWhile :: (a -> Bool) -> [a] -> [a]
```

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`.

```
λ> dropWhile (< 10) [1..20]
[10,11,12,13,14,15,16,17,18,19,20]
λ> dropWhile odd ([1,3..10] ++ [1..10])
[2,3,4,5,6,7,8,9,10]
λ> dropWhile even [1..10]
[1,2,3,4,5,6,7,8,9,10]
λ> dropWhile (> 0) (map (`mod` 5) [1..10])
[0,1,2,3,4,0]
λ> dropWhile (< 3) (takeWhile (< 6) [1..10])
[3,4,5]
```

Dropping with a predicate

```
Data.List.dropWhile :: (a -> Bool) -> [a] -> [a]
```

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`.

```
dropWhile1 :: (a -> Bool) -> [a] -> [a]
dropWhile1 _ [] = []
dropWhile1 p (x:xs)
  | p x         = dropWhile1 p xs
  | otherwise   = x:xs

dropWhile2 :: (a -> Bool) -> [a] -> [a]
dropWhile2 _ [] = []
dropWhile2 p xs@(x:xs')
  | p x        = dropWhile2 p xs'
  | otherwise = xs
```

Iterating

```
Data.List.iterate :: (a -> a) -> a -> [a]
```

`iterate` creates an infinite list where the first item is calculated by applying the function on the second argument, the second item by applying the function on the previous result, and so on.

```
λ> iterate (\x -> x+1) 1 -- equiv iterate (+1) 1
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...
^C Interrupted.
λ> take 10 (iterate (\x -> x+1) 1)
[1,2,3,4,5,6,7,8,9,10]
λ> take 10 (iterate (+1) 1)
[1,2,3,4,5,6,7,8,9,10]
λ> takeWhile (< 10) (iterate (+1) 1)
[1,2,3,4,5,6,7,8,9]
```

Iterating

```
Data.List.iterate :: (a -> a) -> a -> [a]
```

```
iterate1 :: (a -> a) -> a -> [a]
iterate1 f x = let y = f x in y:iterate1 f y
```

```
iterate1 f x
  = x:iterate1 (f x)
  = x:f x:iterate1 (f (f x))
  = x:f x:f (f x):iterate1 (f (f (f x)))
  = ...
```

Iterating

```
Data.List.iterate :: (a -> a) -> a -> [a]
```

```
iterate2 :: (a -> a) -> a -> [a]
iterate2 f x = x:[f y | y <- iterate2 f x]
```

```
iterate2 f x
  = x:[f y | y <- iterate2 f x]
  = x:f x:[f y | y <- iterate2 f (f x)]
  = x:f x:f (f x):[f y | y <- iterate2 f (f (f x))]
  = ...
```

Zipping with functions

```haskell
Data.List.zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

`zipWith` generalises `zip` by zipping with the function given as the first argument, instead of a tupling function.

```haskell
λ> zipWith (+) [0..4] [10..14]
[10,12,14,16,18]
λ> zipWith (\x y -> (x,y)) [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
λ> zipWith (,) [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
λ> f x b = if b then x*10 else x
λ> zipWith f [1,2,3,4] [True,False,True,False]
[10,2,30,4]
```

Zipping with functions

```
Data.List.zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith1 :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith1 _ []     _      = []
zipWith1 _ _      []     = []
zipWith1 f (x:xs) (y:ys) = f x y:zipWith1 f xs ys

zip2 :: [a] -> [b] -> [(a,b)]
zip2 = zipWith1 (,)
```

Zipping with functions – In practice

Determine whether a list is in non-decreasing order.

```
nonDec1 :: Ord a => [a] -> Bool
nonDec1 []           = True
nonDec1 [_]          = True
nonDec1 (x1:x2:xs) = x1 <= x2 && nonDec1 (x2:xs)

nonDec2 :: Ord a => [a] -> Bool
nonDec2 []                 = True
nonDec2 [_]                = True
nonDec2 (x1:xs@(x2:_)) = x1 <= x2 && nonDec2 xs

nonDec3 :: Ord a => [a] -> Bool
nonDec3 xs = and $ zipWith (<=) xs (tail xs)
```

Zipping with functions – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

Zipping with functions – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \qquad M' = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
m = [[1,2],
     [3,4],
     [5,6]]
```

```
m' = [[1,2,0,0,0,0,0],
      [3,4,0,0,0,0,0],
      [5,6,0,0,0,0,0]]
```

Zipping with functions – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

```
λ > m = [[1,2],[3,4],[5,6]]
λ > addExtraColumns 0 m
[[1,2],[3,4],[5,6]]
λ > addExtraColumns 1 m
[[1,2,0],[3,4,0],[5,6,0]]
λ > addExtraColumns 5 m
[[1,2,0,0,0,0,0],[3,4,0,0,0,0,0],[5,6,0,0,0,0,0]]
```

Zipping with functions – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

```haskell
addExtraColumns1 :: Num a => Int -> [[a]] -> [[a]]
addExtraColumns1 k xss = map (++ yss) xss
  where
    yss = replicate k 0

addExtraColumns2 :: Num a => Int -> [[a]] -> [[a]]
addExtraColumns2 k xss = zipWith (++) xss yss
  where
    yss = repeat $ replicate k 0
```

Zipping with functions – In practice

The Leibniz formula for π, named after Gottfried Leibniz, states that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

Zipping with functions – In practice

The Leibniz formula for π, named after Gottfried Leibniz, states that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

```
approxPi1 k = 4 * sum (take k xs)
  where
    ss = [(-1)^n | n <- [0..]]
    xs = zipWith (*) ss (map (1/) (iterate (+2) 1))

approxPi2 k = 4 * sum (take k xs)
  where
    ss = 1:[(-1)*s | s <- ss]
    xs = zipWith (*) ss (map (1/) (iterate (+2) 1))
```

Zipping with functions – In practice

The Leibniz formula for π, named after Gottfried Leibniz, states that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

```
λ> pi
3.141592653589793
λ> let n = 10 in approxPi1 n
3.0418396189294032
λ> let n = 100 in approxPi1 n
3.1315929035585537
λ> let n = 10000 in approxPi1 n
3.1414926535900345
```

Zipping with functions – In practice

The Leibniz formula for π, named after Gottfried Leibniz, states that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

```
λ> ns = iterate (*10) 1
λ> mapM_ print (take 8 [pi / approxPi1 n | n <- ns])
0.7853981633974483
1.0327936535639899
1.003193183258315
1.0003184111600008
1.0000318320017856
1.0000031831090173
1.0000003183099935
1.00000003183099
```

η-conversion

An eta conversion (also written η-conversion) is adding or dropping of abstraction over a function.

The following two values are equivalent under η-conversion:

```
\x -> someFunction x
```

and

```
someFunction
```

Converting from the first to the second would constitute an η-reduction, and moving from the second to the first would be an *eta*-expansion.

The term η-conversion can refer to the process in either direction.

η-conversion

```haskell
insertEntry :: Entry -> AddressBook -> AddressBook
insertEntry entry book = Cons entry book
```

↓ η-reduction

```haskell
insertEntry :: Entry -> AddressBook -> AddressBook
insertEntry entry = Cons entry
```

↓ η-reduction

```haskell
insertEntry :: Entry -> AddressBook -> AddressBook
insertEntry entry = Cons
```

The composition operator

The high-order library operator `.` returns the composition of two function as a single function

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

`f . g`, which is read as `f` composed with `g`, is the function that takes an argument `x`, applies the function `g` to this argument, and applies the function `f` to the result.

The composition operator

Composition can be used to simplify nested function applications, by reducing parentheses ans avoiding the need to explicitly refer to the initial argument.

The composition operator

Composition can be used to simplify nested function applications, by reducing parentheses ans avoiding the need to explicitly refer to the initial argument.

```
odd1 :: Integral a => a -> Bool
odd1 n = not (even n)

odd2 :: Integral a => a -> Bool
odd2 n = (not . even) n   -- i.e., odd2 = \x -> not (even n

odd3 :: Integral a => a -> Bool
odd3 = not . even
```

The composition operator

Composition can be used to simplify nested function applications, by reducing parentheses ans avoiding the need to explicitly refer to the initial argument.

```haskell
twice1 :: (a -> a) -> a -> a
twice1 f x = f (f x)

twice2 :: (a -> a) -> a -> a
twice2 f x = (f . f) x    -- i.e., twice2 = \x -> f (f x)

twice3 :: (a -> a) -> a -> a
twice3 f = f . f
```

The composition operator

Composition is associative

```
f . (g . h) = f . g . h
```

for any functions `f`, `g` and `h` of the appropriate types.

```
sumSqrEven1 :: Integral a => [a] -> a
sumSqrEven1 xs = sum (map (^2) (filter even xs))

sumSqrEven2 :: Integral a => [a] -> a
sumSqrEven2 xs = (sum . map (^2) . filter even) xs

sumSqrEven3 :: Integral a => [a] -> a
sumSqrEven3 = sum . map (^2) . filter even
```

The composition operator

Composition also has an identity, given by the identity function:

```
id :: a -> a
id = \x -> x
```

For any function `f`:

```
id . f  = f
f  . id = f
```

The composition operator

Composition also has an identity, given by the identity function:

```
λ> f = head . id
λ> f [1,2,3,4]
1
```

```
f = head . id
  = \x -> head (id x)
  = \x -> head x
  = head
```

The composition operator

Composition also has an identity, given by the identity function:

```
λ> g = id . head
λ> g [1,2,3,4]
1
```

```
g = id . head
  = \x -> id (head x)
  = \x -> head x
  = head
```

The composition operator

Composition also has an identity, given by the identity function:

```
λ> :type take
take :: Int -> [a] -> [a]
λ> f = take . id
λ> f 3 [1..10]
[1,2,3]
```

```
f = take . id
  = \x -> take (id x)
  = \x -> take x -- :: Int -> ([a] -> [a])
  = take
```

The composition operator

Composition also has an identity, given by the identity function:

```
λ> :type take
take :: Int -> [a] -> [a]
λ> g = id . take
λ> g 3 [1..10]
[1,2,3]
```

```
g = id . take
  = \x -> id (take x)
  = \x -> take x -- :: Int -> ([a] -> [a])
  = take
```

The function application operator

The $ is an operator for function application.

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

All this does is apply a function. So, `f $ x` exactly equivalent to `f x`:

```
λ> head $ [1,2,3,4]
1

λ> tail $ [1,2,3,4]
[2,3,4]

λ> map (+ 1) $ [1,2,3,4]
[2,3,4,5]
```

The function application operator

This seems utterly pointless, until you look beyond the type.

```
λ> :info ($)
($) :: (a -> b) -> a -> b  -- Defined in ‘GHC.Base’
infixr 0 $
```

The function application operator

This seems utterly pointless, until you look beyond the type.

```
λ> :info ($)
($) :: (a -> b) -> a -> b  -- Defined in ‘GHC.Base’
infixr 0 $
```

This little note holds the key to understanding the ubiquity of `($)`: `infixr 0`.

- `infixr` tells us it's an infix operator with right associativity.
- `0` tells us it has the lowest precedence possible.

In contrast, normal function application (via white space)

- is left associative and
- has the highest precedence possible (`10`).

The function application operator

Compare

```
λ> take 10 "Haskell " ++ "rocks!"
"Haskell rocks!"
λ> (take 10  "Haskell ") ++ "rocks!"
"Haskell rocks!"
```

with

```
λ> take 10 $ "Haskell " ++ "rocks!"
"Haskell ro"
λ> take 10 ("Haskell " ++ "rocks!")
"Haskell ro"
```

The function application operator

One pattern where you see the dollar sign used sometimes is between a chain of composed functions and an argument being passed to (the first of) those.

```
λ > sum . drop 3 . take 5 [1..10]
error.

λ > sum . drop 3 . take 5 $ [1..10]
9

λ > (sum . drop 3 . take 5) [1..10]
9

λ > sum . drop 3 $ take 5 [1..10]
9
```

The function application operator

Function application.

```
λ> map (\f -> f 2) [(* i) | i <- [1,2,3,4,5]]
[2,4,6,8,10]

λ> map 2 [(* i) | i <- [1,2,3,4,5]]
error.

λ> map ($ 2) [(* i) | i <- [1,2,3,4,5]]
[2,4,6,8,10]

λ> map ($ 2) [f i | f <- [(*),(+)], i <- [1,2,3,4,5]]
[2,4,6,8,10,3,4,5,6,7]
```

And a curiosity

$ is just an identity function for . . . functions.

```
($) :: (a -> b) -> a -> b
    :: (a -> b) -> (a -> b)

id  :: a -> a
    :: (a -> b) -> (a -> b) -- for a ~ a -> b
```

And a curiosity

$ is just an identity function for . . . functions.

```
($) :: (a -> b) -> a -> b
    :: (a -> b) -> (a -> b)

id  :: a -> a
    :: (a -> b) -> (a -> b) -- for a ~ a -> b
```

```
λ> (sum . drop 3 . take 5) [1..10]
9

λ> sum . drop 3 $ take 5 [1..10]
9

λ> (sum . drop 3) `id` take 5 [1..10]
9

λ> id (sum . drop 3) (take 5 [1..10])
9
```

Origami programming

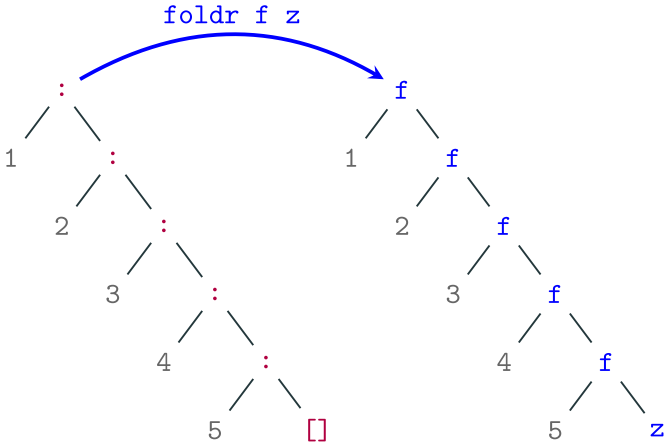
Folding

- In functional programming, `fold` is a family of higher order functions that process a data structure in some order and build a return value.
- This is as opposed to the family of `unfold` functions which take a starting value and apply it to a function to generate a data structure.
- A `fold` deals with two things:
  1. a combining function, and
  2. a data structure.

  The `fold` then proceeds to combine elements of the data structure using the function in some systematic way.

Folding right

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x:xs) = f x (foldr f z xs)
```



Folding right

```haskell
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f z []     = z
  foldr f z (x:xs) = f x (foldr f z xs)

  foldr (+) 0 [1,2,3,4]
= (+) 1 (foldr (+) 0 [2,3,4]
= (+) 1 ((+) 2 (foldr (+) 0 [3,4])
= (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [4])
= (+) 1 ((+) 2 ((+) 3 ((+) 4 (foldr (+) 0 [])
= (+) 1 ((+) 2 ((+) 3 ((+) 4 0) -- stop recursion
= (+) 1 ((+) 2 ((+) 3 4)
= (+) 1 ((+) 2 7)
= (+) 1 9
= 10
```

Folding right

```haskell
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f z []     = z
  foldr f z (x:xs) = f x (foldr f z xs)

  foldr (:) [] [1,2,3,4]
= (:) 1 (foldr (:) [] [2,3,4]
= (:) 1 ((:) 2 (foldr (:) [] [3,4])
= (:) 1 ((:) 2 ((:) 3 (foldr (:) [] [4])
= (:) 1 ((:) 2 ((:) 3 ((:) 4 (foldr (:) [] [])
= (:) 1 ((:) 2 ((:) 3 ((:) 4 []) -- stop recursion
= (:) 1 ((:) 2 ((:) 3 4:[])
= (:) 1 ((:) 2 3:4:[])
= (:) 1 2:3:4:[]
= 1:2:3:4:[]                     -- [1,2,3,4]
```

Folding right

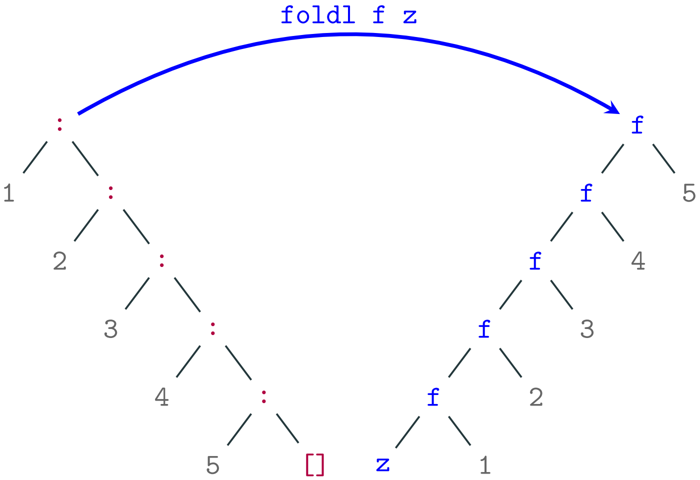
```
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f z []     = z
  foldr f z (x:xs) = f x (foldr f z xs)

  let f x acc = [x]:acc in foldr f [] [1,2,3,4]
= f 1 (foldr f [] [2,3,4]
= f 1 (f 2 (foldr f [] [3,4]))
= f 1 (f 2 (f 3 (foldr f [] [4])))
= f 1 (f 2 (f 3 (f 4 (foldr f [] []))))
= f 1 (f 2 (f 3 (f 4 []))) -- stop recursion
= f 1 (f 2 (f 3 [4]:[]))
= f 1 (f 2 [3]:[4]:[])
= f 1 [2]:[3]:[4]:[]
= [1]:[2]:[3]:[4]:[]        -- [[1],[2],[3],[4]]
```

Folding right

```
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f z []     = z
  foldr f z (x:xs) = f x (foldr f z xs)

  let f x acc = acc ++ [x] in foldr f [] [1,2,3,4]
= f 1 (foldr f [] [2,3,4]
= f 1 (f 2 (foldr f [] [3,4]))
= f 1 (f 2 (f 3 (foldr f [] [4])))
= f 1 (f 2 (f 3 (f 4 (foldr f [] []))))
= f 1 (f 2 (f 3 (f 4 []))) -- stop recursion
= f 1 (f 2 (f 3 ([] ++ [4])))
= f 1 (f 2 ([] ++ [4] ++ [3]))
= f 1 ([] ++ [4] ++ [3] ++ [2])
= [] ++ [4] ++ [3] ++ [2] ++ [1]        -- [4,3,2,1]
```

Folding left

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```



Folding left

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs

  foldl (+) 0 [1,2,3,4]
= foldl (+) ((+) 0 1) [2,3,4]
= foldl (+) ((+) ((+) 0 1) 2) [3,4]
= foldl (+) ((+) ((+) ((+) 0 1) 2) 3) [4]
= foldl (+) ((+) ((+) ((+) ((+) 0 1) 2) 3) 4) []
= ((+) ((+) ((+) ((+) 0 1) 2) 3) 4) -- stop recursion
= ((+) ((+) ((+) 1 2) 3) 4)
= ((+) ((+) 3 3) 4)
= ((+) 6 4)
= 10
```

Folding left

```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldl f z []     = z
  foldl f z (x:xs) = foldl f (f z x) xs

  let fC acc x = x:acc in foldl fC [] [1,2,3,4]
= foldl fC (fC [] 1) [2,3,4]
= foldl fC (fC (fC [] 1) 2) [3,4]
= foldl fC (fC (fC (fC [] 1) 2) 3) [4]
= foldl fC (fC (fC (fC (fC [] 1) 2) 3) 4) []
= (fC (fC (fC (fC [] 1) 2) 3) 4) -- stop recursion
= (fC (fC (fC 1:[] 2) 3) 4)
= (fC (fC 2:1:[] 3) 4)
= (fC 3:2:1:[] 4)
= 4:3:2:1:[]                       -- [4,3,2,1]
```

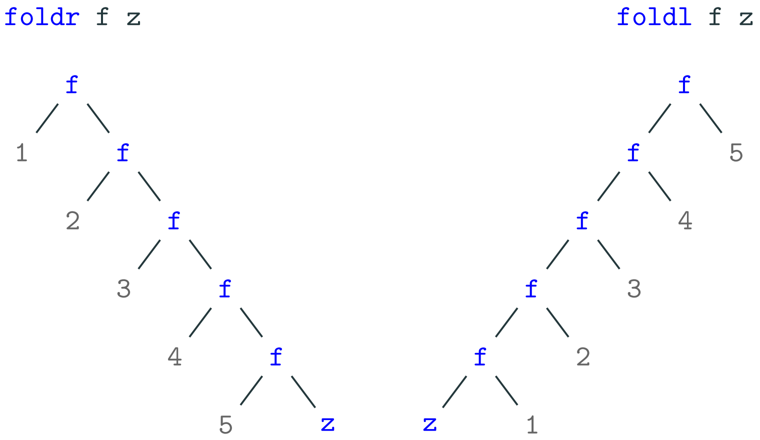
Folding left

```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldl f z []     = z
  foldl f z (x:xs) = foldl f (f z x) xs

  let fC acc x = [x]:acc in foldl fC [] [1,2,3,4]
= foldl fC (fC [] 1) [2,3,4]
= foldl fC (fC (fC [] 1) 2) [3,4]
= foldl fC (fC (fC (fC [] 1) 2) 3) [4]
= foldl fC (fC (fC (fC (fC [] 1) 2) 3) 4) []
= (fC (fC (fC (fC [] 1) 2) 3) 4) -- stop recursion
= (fC (fC (fC [1]:[] 2) 3) 4)
= (fC (fC [2]:[1]:[] 3) 4)
= (fC [3]:[2]:[1]:[] 4)
= [4]:[3]:[2]:[1]:[]             -- [[4],[3],[2],[1]]
```

Folding left

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs

  let fC acc x = acc ++ [x] in foldl fC [] [1,2,3,4]
= foldl fC (fC [] 1) [2,3,4]
= foldl fC (fC (fC [] 1) 2) [3,4]
= foldl fC (fC (fC (fC [] 1) 2) 3) [4]
= foldl fC (fC (fC (fC (fC [] 1) 2) 3) 4) []
= (fC (fC (fC (fC [] 1) 2) 3) 4) -- stop recursion
= (fC (fC (fC []++[1] 2) 3) 4)
= (fC (fC []++[1]++[2] 3) 4)
= (fC []++[1]++[2]++[3] 4)
= []++[1]++[2]++[3]++[4]                -- [1,2,3,4]
```

Folding



Curried functions & friends

Currying

Currying is the process of transforming a function that takes multiple arguments in a tuple as its argument, into a function that takes just a single argument and returns another function which accepts further arguments, one by one, that the original function would receive in the rest of that tuple.

```
f :: a -> b -> c    -- i.e. f :: a -> (b -> c)
```

is the curried form of

```
g :: (a, b) -> c
```

In Haskell, all functions are considered curried: That is, all functions in Haskell take just one argument.

Currying / uncurrying

```haskell
f :: a -> b -> c    -- i.e. f :: a -> (b -> c)

g :: (a, b) -> c
```

You can convert these two types in either directions with the Prelude functions `curry` and `uncurry`:

```haskell
curry :: ((a, b) -> c) -> a -> b -> c

uncurry :: (a -> b -> c) -> (a, b) -> c
```

We have:

```haskell
f = curry g

g = uncurry f
```

Currying / uncurrying

```
f :: a -> b -> c    -- i.e. f :: a -> (b -> c)

g :: (a, b) -> c
```

You can convert these two types in either directions with the Prelude functions `curry` and `uncurry`:

```
curry :: ((a, b) -> c) -> a -> b -> c

uncurry :: (a -> b -> c) -> (a, b) -> c
```

Both forms are equally expressive. It holds:

```
f x y = g (x,y)
```

Uncurrying

```
λ> :type (+)
(+) :: Num a => a -> a -> a

λ> add1 = (+) 1
λ> :type add1
add1 :: Num a => a -> a

λ> add1 2
3

λ> :type uncurry (+)
uncurry (+) :: Num a => (a, a) -> a

λ> uncurry (+) (1,2)
3

λ> uncurry (+) 1
error.
```

Uncurrying

```
λ> zipWith (+) [0..4] [10..14]
[10,12,14,16,18]

λ> :type (+)
(+) :: Num a => a -> a -> a

λ> :type map
map :: (a -> b) -> [a] -> [b]

λ> zip [0..4] [10..14]
[(0,10),(1,11),(2,12),(3,13),(4,14)]

λ> map (\(x,y) -> x+y) $ zip [0..4] [10..14]
[10,12,14,16,18]

λ> map (uncurry (+)) $ zip [0..4] [10..14]
[10,12,14,16,18]
```

Currying

```
λ> :type fst
fst :: (a, b) -> a

λ> fst (1,2)
1

λ> fst 1
error.

λ> type curry fst
curry fst :: a -> b -> a

λ> f = curry fst 1
λ> :type f
f :: Num a => b -> a

λ>f 2
1
```

Currying

```
λ> add p = fst p + snd p
λ> :type add
add :: Num a => (a, a) -> a

λ> add (1,2)
3

λ> add1 = curry add 1
λ> :type add1
add1 :: Num a => a -> a

λ> add1 2
3
```

Flipping

```
flip :: (a -> b -> c) -> b -> a -> c
```

evaluates the function flipping the order of arguments

```
λ> (/) 1 2
0.5

λ> foldr (++) [] ["A","B","C","D"]
"ABCD"

λ> foldr (flip (++)) [] ["A","B","C","D"]
"DCBA"

λ> foldr (:) [] ['a'..'d']
"abcd"

λ> foldr (flip (:)) [] ['a'..'d']
error.
```

Flipping

```
flip :: (a -> b -> c) -> b -> a -> c
```

evaluates the function flipping the order of arguments

```
λ> (/) 1 2
0.5
λ> foldr (++) [] ["A","B","C","D"]
"ABCD"
λ> foldr (flip (++)) [] ["A","B","C","D"]
"DCBA"
λ> foldr (:) [] ['a'..'d']
"abcd"
λ> foldr (flip (:)) [] ['a'..'d']
error.
```

Flipping

```
flip :: (a -> b -> c) -> b -> a -> c
```

evaluates the function flipping the order of arguments

```
flip1 :: (a -> b -> c) -> b -> a -> c
flip1 f x y = f y x
```

```
flip1 :: (a -> b -> c) -> b -> a -> c
flip1 f = \x -> \y -> f y x
```

Flipping – Use cases

```
λ> foldr (:) [] [1..4]
[1,2,3,4]
λ> foldl (flip (:)) [] [1..4]
[4,3,2,1]

λ> foldl (-) 100 [1..4]         -- (((100-1)-2)-3)-4
90
λ> foldr (-) 100 [1..4]         -- 1-(2-(3-(4-100)))
98
λ> foldl (flip (-)) 100 [1..4]  -- 4-(3-(2-(1-100)))
102
λ> foldr (flip (-)) 100 [1..4]  -- (((100-4)-3)-2)-1
90
```

Constant

```haskell
const :: a -> b -> a
```

`const x y` always evaluates to `x`, ignoring its second argument.

```
λ > const 1 2
1
λ > const (2/3) (1/0)
0.6666666666666666
λ > const take drop 5 [1..10]
[1,2,3,4,5]
λ > foldr (\_ acc -> 1 + acc) 0 [1..10]
10
λ > foldr (const (1+)) 0 [1..10]
10
```

Constant

```
const :: a -> b -> a
```

`const x y` always evaluates to `x`, ignoring its second argument.

```
const1 :: a -> b -> a
const1 x _ = x
```

```
const2 :: a -> b -> a
const2 = \x -> \_ -> x
```

Fun with flipping and constant

```haskell
curry id = \x y -> id (x, y)   -- def. curry
         = \x y -> (x, y)      -- def. id
         = \x y -> (,) x y     -- desugar
         = \x -> (,) x         -- eta reduction
         = (,)                 -- eta reduction
```

```
λ> curry id 1 2
(1,2)
λ> (,) 1 2
(1,2)
```

Fun with flipping and constant

```
uncurry const = \(x, y) -> const x y   -- def. uncurry
              = \(x, y) -> x           -- def. const
              = fst                    -- def. fst
```

```
λ> uncurry const (1, 2)
1
λ> fst (1, 2) -- from Data.Tuple (in Prelude)
1
```

Fun with flipping and constant

```
uncurry (flip const)
      = \(x, y) -> (flip const) x y  -- def. uncurry
      = \(x, y) -> const y x         -- def. flip
      = \(x, y) -> y                 -- def. const
      = snd                          -- def. snd
```

```
λ> uncurry (flip const) (1, 2)
2
λ> snd (1, 2) -- from Data.Tuple (in Prelude)
2
```

Fun with flipping and constant

```
uncurry (flip (,))
      = \(x, y) -> (flip (,)) x y  -- def. uncurry
      = \(x, y) -> (,) y x         -- def. flip
      = \(x, y) -> (y, x)          -- desugar
```

```
λ> uncurry (flip (,)) (1, 2)
(2,1)
λ> import Data.Tuple
λ> :type swap
swap :: (a, b) -> (b, a)
λ> swap (1, 2)
(2,1)
```

Processing lists – revisit

Rotations – revisit

Produce all rotations of a list.

```
λ> rotate []
[[]]
λ> rotate [1]
[[1]]
λ> rotate [1,2]
[[2,1],[1,2]]
λ> rotate [1,2,3]
[[3,1,2],[2,3,1],[1,2,3]]
λ> rotate [1,2,3,4]
[[4,1,2,3],[3,4,1,2],[2,3,4,1],[1,2,3,4]]
```

Produce all rotations of a list.

```
shift1xs :: [a] -> [a]
shift1 [] = []
shift1 (x:xs) = xs ++ [x]

rotate3 :: [a] -> [[a]]
rotate3 [] = [[]]
rotate3 xs = foldl (\acc@(xs':acc') _ -> shift xs':acc) [xs
```

Rotations – revisit

Produce all rotations of a list.

```
rotate4 :: [a] -> [[a]]
rotate4 xs = init $ zipWith (++) (tails xs) (inits xs)
-- tails [1,2,3,4] = [[1,2,3,4], [2,3,4], [3,4], [4],
-- inits [1,2,3,4] = [[],        [1],     [1,2], [1,2,3],
```

`Data.List.elem` is the list membership predicate, usually written in infix form, e.g., `` x `elem` xs ``. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

```
-- foldr
elem1 :: (Foldable t, Eq a) => a -> t a -> Bool
elem1 x' xs = foldr f False xs
  where
    f x b = x == x' || b
```

Finding (revisit)

`Data.List.elem` is the list membership predicate, usually written in infix form, e.g., ``x `elem` xs``. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

```haskell
-- eta-reduction
elem2 :: (Foldable t, Eq a) => a -> t a -> Bool
elem2 x' = foldr f False
  where
    f x b = x == x' || b
```

Finding (revisit)

`Data.List.elem` is the list membership predicate, usually written in infix form, e.g., `` x `elem` xs ``. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

```
-- lambda
elem3 :: (Foldable t, Eq a) => a -> t a -> Bool
elem3 x' = foldr (\x b -> x == x' || b) False
```

Filtering (revisit)

`Data.List.filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate.

```
filter3 :: Foldable t => (a -> Bool) -> t a -> [a]
filter3 p xs = foldr f [] xs
  where
   f x acc
     | p x       = x:acc
     | otherwise = acc
```

Repeating (revisit)

`Data.List.repeat` takes an element and returns an infinite list that just has that element.

```
repeat4 :: a -> [a]
repeat4 x = foldr (\_ acc -> x:acc) [] [1..]
```

Repeating (revisi

`Data.Foldable.maximum` returns the maximum value from a list, which must be non-empty, finite, and of an ordered type.

`Data.Foldable.maximum` returns the maximum value from a list, which must be non-empty, finite, and of an ordered type.

```
maximum4 :: Ord a => [a] -> a
maximum4 []     = error "empty list"
maximum4 (x:xs) = foldr f x xs
  where
    f x m = if x > m then x else m

maximum5 :: Ord a => [a] -> a
maximum5 []     = error "empty list"
maximum5 (x:xs) = foldr max x xs
```

`Data.Foldable.maximum` returns the maximum value from a list, which must be non-empty, finite, and of an ordered type.

```
maximum6 :: Ord a => [a] -> a
maximum6 [] = error "empty list"
maximum6 xs = foldl1 max xs

maximum7 :: Ord a => [a] -> a
maximum7 [] = error "empty list"
maximum7 xs = foldr1 max xs
```

Remove duplicate

```
Data.Foldable.nub :: Eq a => [a] -> [a]
```

The `nub` function removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element.

Remove duplicate

```
Data.Foldable.nub :: Eq a => [a] -> [a]
```

The nub function removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element.

```
nub1 :: Eq a => [a] -> [a]
nub1 [] = []
nub1 (x : xs) = x:nub1 (filter (\y -> x/=y) xs)

nub2 :: Eq a => [a] -> [a]
nub2 [] = []
nub2 (x : xs) = x:nub1 xs'
  where
    xs' = filter (/=x) xs
```

Remove duplicate

```
Data.Foldable.nubBy :: (a -> a -> Bool) -> [a] -> [a]
```

The `nubBy` function behaves just like `nub`, except it uses a user-supplied equality predicate instead of the overloaded == function.

Remove duplicate

```
Data.Foldable.nubBy :: (a -> a -> Bool) -> [a] -> [a]
```

The `nubBy` function behaves just like `nub`, except it uses a user-supplied equality predicate instead of the overloaded == function.

```
nubBy1 :: Eq a => (a -> a -> Bool) -> [a] -> [a]
nubBy1 _ [] = []
nubBy1 p (x : xs) = x:nub1 xs'
  where
    xs' = filter (not . p x) xs

nub3 :: Eq a => [a] -> [a]
nub3 = nubBy (==)
```

Remove duplicate

```haskell
Data.Foldable.nubBy :: (a -> a -> Bool) -> [a] -> [a]
```

The `nubBy` function behaves just like `nub`, except it uses a user-supplied equality predicate instead of the overloaded == function.

```haskell
elemBy :: (a -> a -> Bool) -> a -> [a] -> Bool
elemBy _  _ []       =  False
elemBy eq y (x:xs) =  x `eq` y || elemBy eq y xs
nubBy2 :: (a -> a -> Bool) -> [a] -> [a]
nubBy2 eq xs = go xs []
  where
    go []     _          = []
    go (y:ys) xs
      | elemBy eq y xs = go ys xs
      | otherwise      = y:go ys (y:xs)
```