Functional programming

Lecture 03 – Lists

(version: 2026-02-26–11:54:55)

Stéphane Vialette
stephane.vialette@univ-eiffel.fr

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

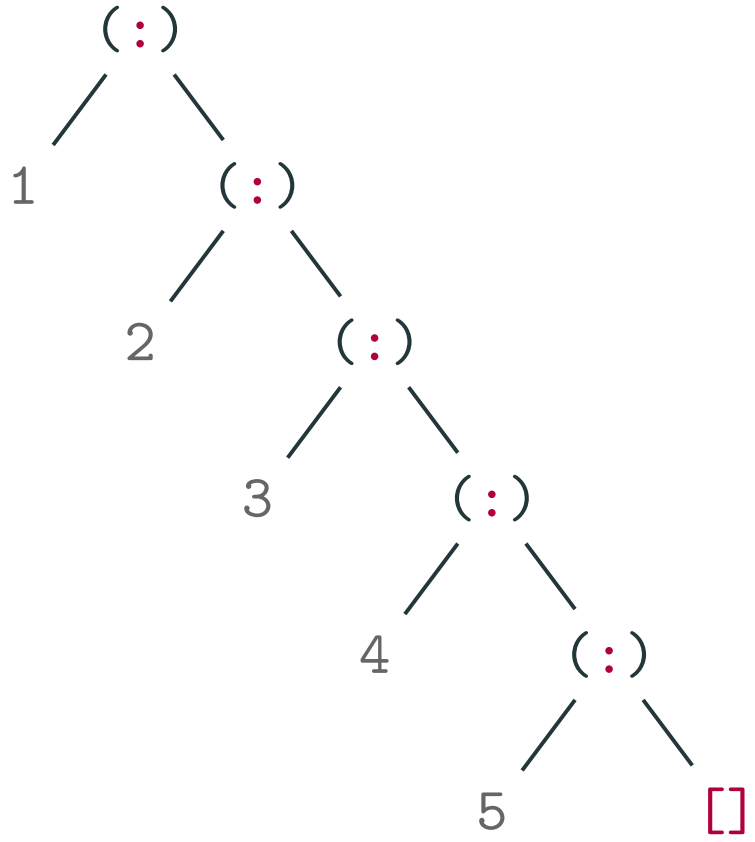
```
[]                                  :: [a]

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

head2 :: [a] -> 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] -> [a]
tail1 []       = error "*** Exception: tail: empty list"
tail1 (x : xs) = xs

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

Enumerations

Enumerating lists of integers

```
-- List of numbers 1,2,...,10.
[1..10]

-- Infinite list of numbers 1,2,...
[1..]

-- Empty list; ranges only go forwards.
[10..1]

-- Negative integers.
[0,-1..]

-- List from 1 to 10 by 2 = [1,3,5,7,9]
[1,3..10]

-- List from -1 to 10 by 4 = [-1,3,7]
[-1,3..10]
```

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 > [1..]
[1,2,3,4,5,6,7,8,9,... ^CInterrupted.
λ > xs = [1..]
λ > head xs
1
λ > head (tail xs)
2
λ > tail xs
[2,3,4,5,6,7,8,9,... ^CInterrupted.
```

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.
λ > [10,0..0]
[10,0]
λ > [100,90..0]
[100,90,80,70,60,50,40,30,20,10,0]
```

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]
[5.0,4.333333333333333,3.6666666666666666,2.9999999999999999]
```

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 characters

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

Enumerating lists of characters

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'
λ> ['a',succ 'a', succ (succ 'a'), succ (succ (succ 'a'))]
"abcd"
```

Enumerating lists of characters

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'
λ> ['A',succ 'A', succ (succ 'A'), succ (succ (succ 'A'))]
"ABCD"
```

Enumerating lists of characters

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 characters

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 characters

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 > fromEnum 'A'
65
λ > toEnum 65 :: Char
'A'
λ > [fromEnum c | c <- "ABCD"]
[65,66,67,68]
λ > [toEnum i :: Char | i <- [65,66,67,68]]
"ABCD"
```

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.

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

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

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

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

```
import itertools

(x*x for x in itertools.count(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,
 14.285714285714286,12.5,11.111111111111,10.0]
```

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]]]
```

Infinite lists

```
λ > let xs = [] in [x | x <- xs] == xs
True
λ > let xs = [1..1_000_000] in [x | x <- xs] == xs
True
λ > let xs = [1..] in [x | x <- xs] == xs
^CInterrupted.
λ > let xs = [1..] in [x | x <- tail xs] == xs
False
```

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]`.

```
-- assuming we don't know anything about Data.Foldable.sum
-- sums n = [sum [1..k] | k <- [1..n]]
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 anything about Data.Foldable.sum
-- sumsSq n = (map (sum . map (^2)) . tail . inits) [1..n]
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.

```
p2s :: Num a => [a]
p2s = [2*p2 | p2 <- 1 : p2s]
```

```
λ > p2s
[2,4,8,16,32,64,128,256,512,1024,2048... ^CInterrupted.
λ > take 11 p2s
[2,4,8,16,32,64,128,256,512,1024,2048]
λ > head (drop 120 p2s)
2658455991569831745807614120560689152
```

Problem solving with list comprehensions

Compute the infinite list of the powers of 2.

```
p2s :: Num a => [a]
p2s = [2*p2 | p2 <- 1 : p2s]
```

```
1 : 2*1 : p2s
```

```
1 : 2*1 : 2*2*1 : p2s
```

```
1 : 2*1 : 2*2*1 : 2*2*2*1 : p2s
```

```
1 : 2*1 : 2*2*1 : 2*2*2*1 : 2*2*2*2*1 : p2s
```

```
...
```

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"
```

Problem solving with list comprehensions

Exercice

Compute the infinite list of all binary strings (think about).

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

```
λ > take 6 binaries'
["0","00","000","0000","00000","000000"]
λ > head (drop 20 binaries')
"00000000000000000000"
```

Problem solving with list comprehensions

Exercice

A positive integer is perfect if it equals the sum of its divisors, excluding the number itself. Using list comprehensions define the two functions

```
divisors :: Int -> [Int]
perfects :: Int -> [Int]
```

that returns the list of all proper divisors of a given integer (function `divisors`) and the list of all perfect numbers up to a given limit function `perfects`. For example:

```
λ > [divisors n | n <- [1..10]]
[[],[1],[1],[1,2],[1],[1,2,3],[1],[1,2,4],[1,3],[1,2,5]]
λ > perfects 500
[6,28,496]
```

Problem solving with list comprehensions

Exercice

Consider, the following session

```
λ > [x | x <- [1..10], even x]
[2,4,6,8,10]
λ > [x | x <- [1..], x <= 10 && even x]
[2,4,6,8,10
^C Interrupted.
λ > [x | x <- [1..], even x && x <= 10]
[2,4,6,8,10
^C Interrupted.
```

Comment.

Processing lists – basic functions (toolbox)

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] -- == elem 2 [1..5]
True
λ> 8 `elem` [1..5] -- == elem 8 [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

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

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

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

Repeating

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

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

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

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

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

repeat4 :: Enum a => a -> [a]
repeat4 x = [x,x..]
```

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..]   -- infinite list
[1,2,3,4,5,6,7,8,9,10]
λ > take 20 [1..10] -- short list
[1,2,3,4,5,6,7,8,9,10]
λ > take 0 [1..]
[]
λ > take (-1) [1..] -- negative integer
[]
```

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..]     -- infinite list
[11,12,13,14,15,16,17,18,19,20,...
^C Interrupted.
λ > drop (-1) [1..4] -- negative integer
[1,2,3,4]
λ > drop 20 [1..10]  -- short list
[]
```

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 0 xs         = xs
rotate1 _ []         = []
rotate1 n (x : xs) = rotate1 (n-1) (xs ++ [x])

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

Accumulating

Exercice

Is this implementation correct ?

```
rotate3 :: Int -> [a] -> [a]
rotate3 = 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
```

Hint: it is not !

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 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)

replicate3 :: (Ord t, Num t) => t -> a -> [a]
replicate3 n = take n . repeat
```

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'
```

Rotating

Exercice

define the function

```
tails' :: [a] -> [[a]]
```

that returns all final segments of the argument, shortest first.

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

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
```

Reversing

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

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

We shall see soon that an implementation for `reverse` can be both short and efficient.

```
reverse3 :: [a] -> [a]
reverse3 = foldl (flip (:)) []
```

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

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 : init1 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 > take 4 (inits [1..])
[[],[1],[1,2],[1,2,3]]
```

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 y xs
```

Concatening

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

`concat` concatenates a list of lists.

Concatening

```
Data.List.concat :: [[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 :: [[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.

```
zip1 :: [a] -> [b] -> [(a, b)]
zip1 []       _        = []
zip1 _        []       = []
zip1 (x : xs) (y : ys) = (x,y) : zip1 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.

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

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

index3 :: Enum a => a -> [b] -> [(a, b)]
index3 n = zip [n..] -- eta reduction
```

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`.

```haskell
-- 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

```
and :: [Bool] -> Bool
```

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

Anding

```
and :: [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

```
and :: [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 :  _) = False
and1 ( True : bs) = and1 bs

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

Oring

```
or :: [Bool] -> Bool
```

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

Oring

```
or :: [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

```
or :: [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 :  _) = True
or1 (False : bs) = or1 bs

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

Maximizing

```
maximum :: [a] -> a
```

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

```
maximum :: [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 x

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

λ > :type max
max :: Ord a => a -> a -> a
```

Maximizing

```haskell
maximum :: [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'
```

Deleting

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

`delete` removes the first occurrence of the specified element from its list argument.

```
λ > delete 1 [1..10]
[2,3,4,5,6,7,8,9,10]
λ > delete 5 [1..10]
[1,2,3,4,6,7,8,9,10]
λ > delete 11 [1..10]
[1,2,3,4,5,6,7,8,9,10]
λ > delete 3 [1,1,2,2,3,3,4,4,5,5]
[1,1,2,2,3,4,4,5,5]
```

Deleting

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

delete1 :: Eq a => a -> [a] -> [a]
delete1 _ [] = []
delete1 x (x' : xs) = if x == x'
                      then xs
                      else x' : delete x xs

delete2 :: Eq a => a -> [a] -> [a]
delete2 x [] = []
delete2 x (x' : xs)
  | x == x'    = xs
  | otherwise = delete2 x xs
```

Deleting

Write the function `deleteAll :: Ord a => [a] -> [a]` that removes all occurrence of the specified element from its list argument.

```
λ > deleteAll 0 (intersperse 0 [1..9])
[1,2,3,4,5,6,7,8,9]
λ > deleteAll 10 (intersperse 0 [1..9])
[1,0,2,0,3,0,4,0,5,0,6,0,7,0,8,0,9]
```

Sorting

```haskell
sort :: Ord a => [a] -> [a]
```

Sorting

```
sort :: Ord a => [a] -> [a]

λ > sort []
[]
λ > sort [1,4,2,3,5,1,3,2,5,4]
[1,1,2,2,3,3,4,4,5,5]
λ > let n = 10000 in sort [1..n] == [1..n]
True
```

Sorting

```
sort :: Ord a => [a] -> [a]

sort1 :: Ord a => [a] -> [a]
sort1 []       = []
sort1 (x : xs) = sort1 ys ++ [x] ++ sort1 zs
    where
      ys = [y | y <- xs, y <= x]
      zs = [z | z <- xs, z >  x]
```

Sorting

```
sort :: Ord a => [a] -> [a]

sort2 :: Ord a => [a] -> [a]
sort2 []  = []
sort2 [x] = [x]
sort2 xs  = let (ys, zs) = split2 ([], []) xs
            in merge (sort2 ys) (sort2 zs)
  where
    split2 yzs       []             = yzs
    split2 (ys, zs) [x]             = (x : ys, zs)
    split2 (ys, zs) (x : x' : xs) = split2 (x : ys, x' : zs) xs

    merge []       zs = zs
    merge ys       [] = ys
    merge (y : ys) (z : zs)
      | y <= z        = y : merge ys       (z : zs)
      | otherwise     = z : merge (y : ys) zs
```

Sorting

```haskell
sort :: Ord a => [a] -> [a]

-- incognito foldr ;-) !
sort3 :: Ord a => [a] -> [a]
sort3 = go []
  where
    go acc []       = acc
    go acc (x : xs) = bubble x (go acc xs)
      where
        bubble x []    = [x]
        bubble x (y : xs)
          | x <= y     = x : bubble y xs
          | otherwise = y : bubble x xs
```

Sorting

```haskell
sort :: Ord a => [a] -> [a]

-- simple but inefficient
sort4 :: Ord a => [a] -> [a]
sort4 [] = []
sort4 xs = let (x, xs') = extractMin xs in x : sort4 xs'
  where
    extractMin xs = let x = minimum xs in (x, delete x xs)
```