Week 1A: Basics of Functional Programming

Haskell Basics

First Steps

Description:

> Write a function called `addAndDouble` which adds two numbers together and then doubles the result, by replacing `undefined` with the proper expression.
>
> Examples:
>
> - `addAndDouble 1 1 = 4`
> - `addAndDouble 14 7 = 42`

Solution:

```haskell
module Solution where

import Library

addAndDouble x y = 2 * (x+y)
```

Spec tests:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_addAndDouble_example :: Property
prop_addAndDouble_example = addAndDouble 1 1 === 4

prop_addAndDouble_correct :: Int -> Int -> Property
prop_addAndDouble_correct x y = addAndDouble x y === 2 * (x+y)
```

Fix the syntax

> The given script contains three syntactic errors. Correct these errors and check that the value of `n` is computed correctly.

Solution:

```
module Solution where

import Library

n = a `div` length xs
  where
    a = 10
    xs = [1,2,3,4,5]
```

Spec tests: N/A

Any definition will do

> Write down definitions that have the following types; it does not matter what the definitions actually do as long as their types are correct.
>
> ```
> bools :: [Bool]
> add :: Int -> Int -> Int -> Int
> copy :: a -> (a,a)
> choose :: Bool -> a -> a -> a
> ```

Solution:

```
module Solution where

import Library

bools :: [Bool]
bools = [True,False]

add :: Int -> Int -> Int -> Int
add x y z = x + y + z

copy :: a -> (a,a)
copy x = (x,x)

choose :: Bool -> a -> a -> a
choose b x y = if b then x else y
```

Spec tests:

```
module Test where

import Test.QuickCheck
```

```haskell
import Library
import Solution

prop_bool = total (bools :: [Bool])

prop_add = total (add :: Int -> Int -> Int -> Int)

prop_copy_bool = total (copy :: Bool -> (Bool,Bool))

prop_copy_int = total (copy :: Int -> (Int,Int))

prop_choose_bool = total (Solution.choose :: Bool -> Bool -> Bool -> Bool)

prop_choose_int = total (Solution.choose :: Bool -> Int -> Int -> Int)
```

Reverse Quicksort

> Modify the definition of the function `qsort` , which implements the *quicksort* algorithm so that is produces a *reverse* sorted version of the list.

Solution:

```haskell
module Solution where

import Library

qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) =
  qsort larger ++ [x] ++ qsort smaller
    where
      smaller = [ y | y <- xs , y <= x ]
      larger  = [ y | y <- xs , y > x  ]
```

Spec tests:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

import Data.List (sort)

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

```haskell
prop_qsort_length :: [Int] -> Property
prop_qsort_length xs = length (qsort xs) === length xs

prop_qsort_sorted :: [Int] -> Bool
prop_qsort_sorted xs = sorted (qsort xs)

prop_qsort_correct :: [Int] -> Property
prop_qsort_correct xs = qsort xs === reverse (sort xs)
```

Associating the arrows

Q: Which of these types does the type `a -> b -> c -> d` correspond to?

A: `a -> (b -> (c -> d))`

Quadratic equations

> Write a function called `solveQuadratic` that takes in three arguments of type `Double` (`a` , `b` , and `c`) and returns a list consisting of all (real-valued) solutions of the quadratic equation $ax^2+bx+c=0$.
>
> **Hint.** Use a `let` or `where` expression to define the square root of the discriminant $D=b^2-4ac$.
>
> - If $D<0$, there is no solution.
> - If $D=0$, the single solution is $\frac{-b}{2a}$
> - If $D>0$, the two solutions are $\frac{\sqrt{D}-b}{2a}$ and $\frac{-\sqrt{D}-b}{2a}$.
> -
>
> The function for computing the square root in Haskell is called `sqrt` .

Solution:

```haskell
module Solution where

import Library

solveQuadratic :: Double -> Double -> Double -> [Double]
solveQuadratic a b c =
  if d > 0
  then [(-b+d)/(2*a), (-b-d)/(2*a)]
  else if d == 0
       then [-b/(2*a)]
       else []
  where
```

```
    discr = b^2 - 4*a*c
    d = sqrt discr
```

Spec tests:

```
module Test where

import Test.QuickCheck
import Library
import Solution

epsilon = 0.001

prop_solveQuadratic :: Property
prop_solveQuadratic = solveQuadratic 1 0 (-9) === [3,-3]

prop_quadratic_no_solutions :: Positive Integer -> Positive Double -> Positive Doubl
prop_quadratic_no_solutions (Positive n) (Positive a) (Positive c) = solveQuadratic
  where
    b = sqrt (a*c) / (4 + fromInteger n)

prop_quadratic_one_solution :: NonZero Double -> Double -> Property
prop_quadratic_one_solution (NonZero a) b =
        (d == 0) ==> length sols == 1 .&&. abs (head sols - s) < epsilon
  where
    d = b^2 - 4 * a * c
    c = -b^2 / (-4 * a)
    s = -b / (2 * a)
    sols = solveQuadratic a b c

prop_quadratic_two_solutions :: NonZero Double -> Double -> Double -> Property
prop_quadratic_two_solutions (NonZero u) v w =
         (v /= w) ==>
         length sols == 2
    .&&. any (\x -> abs (x - v) < epsilon) sols
    .&&. any (\x -> abs (x - w) < epsilon) sols
  where
    a = u
    b = -u*(v+w)
    c = u*v*w
    sols = solveQuadratic a b c
```

Luhn Algorithm

> The *Luhn algorithm* (Wikipedia) is used to check bank card numbers for simple errors such as mistyping a digit, and proceeds as follows:
>
> - consider each digit as a separate number;

> - moving left, double every other number from the second last;
> - subtract 9 from each number that is now greater than 9;
> - add all the resulting numbers together;
> - if the total is divisible by 10, the card number is valid.
>
> Define a function `luhnDouble :: Int -> Int` that doubles a digit and subtracts 9 if the result is greater than 9. For example:
>
> ```
> > luhnDouble 3
> 6
>
> > luhnDouble 6
> 3
> ```
>
> Using `luhnDouble` and the integer remainder function `mod`, define a function `luhn :: Int -> Int -> Int -> Int -> Bool` that decides if a four-digit bank card number is valid. For example:
>
> ```
> > luhn 1 7 8 4
> True
>
> > luhn 4 7 8 3
> False
> ```
>
> Now define a function `luhnFinal :: Int -> Int -> Int -> Int` that returns the fourth digit of a four-digit bank card number. For example:
>
> ```
> > luhnFinal 1 7 8
> 4
>
> > luhnFinal 4 7 8
> 8
> ```

Solution:

```haskell
module Solution where

import Library

luhnDouble x = if double_x > 9 then double_x - 9 else double_x
  where double_x = 2*x

luhn x y z w = (luhnDouble x + y + luhnDouble z + w) `mod` 10 == 0
```

```haskell
luhnFinal x y z = 10 - (luhnDouble x + y + luhnDouble z) `mod` 10
```

Spec tests:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

digit :: Gen Int
digit = choose (0, 9)

digits :: Gen [Int]
digits = vectorOf 4 digit

luhnDouble_spec x = if double_x > 9 then double_x - 9 else double_x
  where double_x = 2*x

luhn_spec x y z w = (luhnDouble_spec x + y + luhnDouble_spec z + w) `mod` 10 == 0

prop_luhnDouble_correct :: Property
prop_luhnDouble_correct = forAll digit $ \x -> luhnDouble x === luhnDouble_spec x

prop_luhn_correct :: Property
prop_luhn_correct = forAll digits $ \[x,y,z,w] -> luhn x y z w === luhn_spec x y z w

prop_luhnFinal_correct :: Property
prop_luhnFinal_correct = forAll (vectorOf 3 digit) $ \[x,y,z] -> property $ luhn_spe
```

Working with Lists

Half the list it used to be

> Using library functions, define a function `halve :: [a] -> ([a],[a])` that splits an even-lengthed list into two halves. For example:
>
> ```
> > halve [1,2,3,4,5,6]
> ([1,2,3],[4,5,6])
> ```
>
> *Hint*: Some of the following library functions may come in handy:
>
> - `head :: [a] -> a`
> - `tail :: [a] -> [a]`
> - `length :: [a] -> Int`

> - `reverse :: [a] -> [a]`
> - `take :: Int -> [a] -> [a]`
> - `drop :: Int -> [a] -> [a]`
> - `mod :: Int -> Int -> Int`

Solution:

```haskell
module Solution where

import Library

halve xs = (take n xs, drop n xs)
  where
    n = length xs `div` 2
```

Spec tests:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_halve_same_length :: [Int] -> Property
prop_halve_same_length xs = length xs `mod` 2 == 0 && length xs > 0 ==> length xs1 =
  where
    (xs1,xs2) = halve xs

prop_halve_join :: [Int] -> Property
prop_halve_join xs = length xs `mod` 2 == 0 ==> xs == xs1 ++ xs2
  where
    (xs1,xs2) = halve xs

prop_halve_empty :: Bool
prop_halve_empty = length xs1 == 0 && length xs2 == 0
  where
    (xs1,xs2) = halve []
```

Most lists will do

> Write down lists that have the following types; each list should have at least three elements and the elements should all be different from each other.
>
> ```haskell
> nums :: [Int]
> bools :: [[Bool]]
> ```

> ```
> lists :: [[[a]]]
> ```

Solution:

```
module Solution where

import Library

nums :: [Int]
nums = [1,2,3]

bools :: [[Bool]]
bools = [[], [True], [False]]

lists :: [[[a]]]
lists = [ [] , [[]] , [[],[]] ]
```

Spec tests:

```
module Test where

import Test.QuickCheck
import Library
import Solution

import qualified Data.Set as Set

prop_nums_length = length (nums :: [Int]) >= 3

prop_nums_distinct = Set.size (Set.fromList nums) == length nums

prop_bools_length = length (bools :: [[Bool]]) >= 3

prop_bools_distinct = Set.size (Set.fromList bools) == length bools

prop_lists_length = length (lists :: [[[a]]]) >= 3

--prop_lists_distinct = Set.size (Set.fromList (lists :: [[[Bool]]])) == length list
```

Initial fragment

> Implement the function `init` that removes the last element from a non-empty list, either in terms of other library functions or directly.
>
> *Hint*: Some of the following library functions may come in handy:
>
> - `head :: [a] -> a`

> - `tail :: [a] -> [a]`
> - `length :: [a] -> Int`
> - `reverse :: [a] -> [a]`
> - `take :: Int -> [a] -> [a]`
> - `drop :: Int -> [a] -> [a]`
> - `mod :: Int -> Int -> Int`

Solution:

```haskell
module Solution where

import Library

import Prelude hiding (init)

init :: [a] -> [a]
init xs = reverse (tail (reverse xs))
```

Spec tests:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

import Prelude hiding (init)
import qualified Prelude

prop_init :: [Int] -> Property
prop_init xs = not (null xs) ==> init xs == Prelude.init xs
```

Tail, but safer

> Define a function `safeTail :: [a] -> [a]` that behaves in the same way as `tail` on non-empty lists, and returns the empty list when given an empty list.

Solution:

```haskell
module Solution where

import Library

safeTail [] = []
```

```haskell
safeTail xs = tail xs
```

Spec tests:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_safeTail_empty :: Bool
prop_safeTail_empty = safeTail ([] :: [Int]) == []

prop_safeTail_nonempty :: [Int] -> Property
prop_safeTail_nonempty xs = not (null xs) ==> safeTail xs == tail xs
```

Counting down fast

> Write a function `countDownBy5` that given a particular number, gives a list starting there and counting down by 5 until you get to 0.
>
> Examples:
>
> - `countDownBy5 4 = [4]`
> - `countDownBy5 5 = [5,0]`
> - `countDownBy5 12 = [12,7,2]`

Solution:

```haskell
module Solution where

import Library

countDownBy5 :: Int -> [Int]
countDownBy5 n = [n,n-5..0]
```

Spec test:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_countDownBy5_correct :: Int -> Property
```

```haskell
prop_countDownBy5_correct n = countDownBy5 n === [n,n-5..0]
```

Removal

> Implement a function `remove :: Int -> [a] -> [a]` which takes a number `n` and a list and removes the element at position `n` from the list. For example:
>
> ```
> > remove 1 [1,2,3,4]
> [1,3,4]
> ```
>
> **Hint.** Make use of the library functions `take` and `drop` .

Solution:

```haskell
module Solution where

import Library

remove n xs = take n xs ++ drop (n+1) xs
```

Spec test:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_remove_correct :: [Int] -> Int -> [Int] -> Property
prop_remove_correct xs y zs = remove (length xs) (xs ++ [y] ++ zs) === xs ++ zs
```

Triplets

> Implement a function `triplets :: [a] -> [(a,a,a)]` that computes the list of all triplets of adjacent values in the list. For example:
>
> ```
> > triplets [1,2,7,4,5]
> [(1,2,7),(2,7,4),(7,4,5)]
> > triplets [6,42]
> []
> > triplets "qwerty"
> [('q','w','e'),('w','e','r'),('e','r','t'),('r','t','y')]
> ```

> **Note.** This was a sub-question on the exam of 16/4/2021.

Solution:

```
module Solution where

import Library

triplets :: [a] -> [(a,a,a)]
triplets xs = zip3 xs (drop 1 xs) (drop 2 xs)
```

Spec test:

```
module Test where

import Test.QuickCheck
import Library
import Solution

prop_triplets_example :: Bool
prop_triplets_example = triplets [1,2,3,4,5] == [(1,2,3),(2,3,4),(3,4,5)]

triplets_spec :: [a] -> [(a,a,a)]
triplets_spec xs = zip3 xs (drop 1 xs) (drop 2 xs)

prop_triplets_correct :: [Int] -> Property
prop_triplets_correct xs = triplets xs === triplets_spec xs
```

List comprehensions

Evens

> Using a list comprehension, define a function that selects all the **even** numbers from a list.
>
> Example:
>
> - `evens [1..10] = [2,4,6,8,10]`

Solution:

```haskell
module Solution where

import Library

evens :: [Int] -> [Int]
evens xs = [ x | x <- xs, even x ]
```

Spec test:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_evens_correct :: [Int] -> Property
prop_evens_correct xs = evens xs === filter even xs
```

Sum of Squares

> Using a list comprehension and the library function `sum :: [Int] -> Int` , define a function `sumOfSquares :: Int -> Int` that when given a positive integer `n` computes the sum `1^2 + 2^2 + 3^2 + ... + n^2` of the first `n` squares`.

Solution:

```haskell
module Solution where

import Library

sumOfSquares n = sum [ i*i | i <- [1..n] ]
```

Spec test:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_sumOfSquares :: Positive Int -> Bool
prop_sumOfSquares (Positive n) = 6 * (sumOfSquares n) == n * (2 * n*n + 3 * n + 1)
```

Replication

> Using a list comprehension, redefine the library function `replicate :: Int -> a -> [a]` that produces a list of identical elements. For example:
>
> ```
> > replicate 3 True
> [True,True,True]
> ```

Solution:

```haskell
module Solution where

import Library

import Prelude hiding (replicate)

replicate n x = [ x | _ <- [1..n] ]
```

Spec tests:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

import Prelude hiding (replicate)

prop_replicate_zero :: Int -> Property
prop_replicate_zero x = replicate 0 x === []

prop_replicate_length :: NonNegative Int -> Int -> Property
prop_replicate_length (NonNegative n) x = length (replicate n x) === n

prop_replicate_elems :: Int -> Int -> Bool
prop_replicate_elems n x = all (== x) $ replicate n x
```

Pythagorean

> A triple `(x,y,z)` of positive integers is *Pytagorean* if it satisfies the equation `x^2 + y^2 = z^2`. Using a list comprehension with three generators, define a function `pyths :: Int -> [(Int,Int,Int)]` that returns the list of all such triples whose components are at most a given limit. For example:
>
> ```
> > pyths 10
> [(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
> ```

Solution:

```
module Solution where

import Library

pyths n = [(x,y,z) | x <- [1..n], y <- [1..n], z <- [1..n], x*x + y*y == z*z]
```

Spec tests:

```
module Test where

import Test.QuickCheck
import Library
import Solution

prop_all_pythagorean :: Int -> Bool
prop_all_pythagorean n = all (\(x,y,z) -> x*x + y*y == z*z) $ pyths n

pyths_count :: Int -> Int
pyths_count n = length [ ()
                       | x <- [1..(n-1)], y <- [1..(n-1)]
                       , let z = round (sqrt (fromIntegral (x*x+y*y)))
                       , z <= n
                       , x*x + y*y == z*z
                       ]

prop_no_missing_pyths :: NonNegative Int -> Bool
prop_no_missing_pyths (NonNegative n) = length (pyths n) == pyths_count n
```

Perfect numbers

> A positive integer is *perfect* if it equals the sum of all its factors, excluding the number itself. Using a list comprehension and the function `factors` (already defined in the Library tab), define a function `perfects :: Int -> [Int]` that returns the list of all perfect numbers up to a given limit. For example:
>
> ```
> > perfects 500
> [6,28,496]
> ```

Solution:

```
module Solution where

import Library
```

```haskell
perfects n = [ k | k <- [1..n], sum (factors k) == k ]
```

Spec test:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_all_perfect :: Int -> Bool
prop_all_perfect n = all perfect $ perfects n
  where
    perfect k = sum (factors k) == k

perfects_spec n = [ k | k <- [1..n], sum (factors k) == k ]

prop_no_missing_perfects :: Int -> Bool
prop_no_missing_perfects n = length (perfects n) == length (perfects_spec n)
```

Scalar product

> The *scalar product* of two lists of integers `xs` and `ys` of length n is given by the sum of the products of corresponding integers. For example, the scalar product of `[1,2,3]` and `[4,5,6]` is `1*4 + 2*5 + 3*6 = 32` Define a function `scalarproduct :: [Int] -> [Int] -> Int` that returns the scalar product of two lists. For example:
>
> ```
> > scalarproduct [1,2,3] [4,5,6]
> 32
> ```
>
> Return `0` in case `xs` and `ys` are of different lengths.

Solution:

```haskell
module Solution where

import Library

scalarproduct xs ys
  | length xs == length ys = sum [x * y | (x, y) <- zip xs ys]
  | otherwise = 0
```

Spec tests:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_scalarproduct_empty :: Bool
prop_scalarproduct_empty = scalarproduct [] [] == 0

prop_scalarproduct_cons1 :: Int -> Int -> Int -> Property
prop_scalarproduct_cons1 x y n = forAll (liftArbitrary2 (vector n) (vector n))
                                        (\(xs,ys) ->  scalarproduct (x:xs) (y:ys) ==
                                                      x*y + scalarproduct xs ys)

prop_scalarproduct_diff_lengths :: Int -> Int -> Property
prop_scalarproduct_diff_lengths m n = (m /= n) ==> forAll (liftArbitrary2 (vector m)
                                        (\(xs, ys) ->  scalarproduct (xs::[Int]) ys
```

Divisors

> Implement a function `divisors :: Int -> [Int]` that returns the divisors of a natural number. For example:
>
> ```
> > divisors 15
> [1,3,5,15]
> ```
>
> **Hint.** First implement a function `divides :: Int -> Int -> Bool` that decides if one integer is divisible by another.

Solution:

```haskell
module Solution where

import Library

divides x y = x `mod` y == 0

divisors x = [ y | y <- [1..x], divides x y ]
```

Spec test:

```haskell
module Test where

import Test.QuickCheck
import Library
```

```haskell
import Solution

prop_divisors_in_range :: Positive Int -> Property
prop_divisors_in_range (Positive x) = forAll (elements $ divisors x) $ \y ->
  y > 0 .&&. y <= x

prop_divisors_divide :: Positive Int -> Property
prop_divisors_divide (Positive x) = forAll (elements $ divisors x) $ \y ->
  x `mod` y == 0

prop_divisors_all :: Positive Int -> Property
prop_divisors_all (Positive x) = forAll (chooseInt (1,x)) $ \y ->
  (y `elem` divisors x) === (x `mod` y == 0)
```

What are your coordinates?

> Suppose that a *coordinate grid* of size $m \times n$
>
> is given by the list of all pairs `(x,y)` of integers such that `0 =< x =< m` and `0 =< y =< n`. Using a list comprehension, define a function `grid :: Int -> Int -> [(Int,Int)]` that returns a coordinate grid of a given size. For example:
>
> ```
> > grid 1 2
> [(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]
> ```
>
> Next, using a list comprehension and the function `grid` you just defined, define a function `square :: Int -> [(Int,Int)]` that returns a coordinate square of size `n`, excluding the diagonal from `(0,0)` to `(n,n)`. For example:
>
> ```
> > square 2
> [(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)]
> ```

Solution:

```haskell
module Solution where

import Library

grid m n = [(i,j) | i <- [0..m], j <- [0..n]]

square n = [(i,j) | (i,j) <- grid n n , i /= j]
```

Spec test:

```
module Test where

import Test.QuickCheck
import Library
import Solution

import qualified Data.Set as Set

prop_grid_length :: NonNegative Int -> NonNegative Int -> Bool
prop_grid_length (NonNegative m) (NonNegative n) =
  length (grid m n) == (m+1) * (n+1)

prop_grid_unique :: Int -> Int -> Bool
prop_grid_unique m n = Set.size (Set.fromList g) == length g
  where g = grid m n

prop_grid_in_range :: Int -> Int -> Bool
prop_grid_in_range m n = all (\(i,j) -> i >= 0 && i <= m && j >= 0 && j <= n) $ gric

prop_square_length :: NonNegative Int -> Bool
prop_square_length (NonNegative n) =
  length (square n) == (n+1)*n

prop_square_unique :: Int -> Bool
prop_square_unique n = Set.size (Set.fromList g) == length g
  where g = square n

prop_square_in_range :: Int -> Bool
prop_square_in_range n = all (\(i,j) -> i >= 0 && i <= n && j >= 0 && j <= n) $ squa

prop_square_no_diag :: Int -> Bool
prop_square_no_diag n = all (\(i,j) -> i /= j) $ square n
```

Riffle raffle

> Implement a function `riffle :: [a] -> [a] -> [a]` that takes two lists of the same length and interleaves their elements in alternating order. For example:
>
> ```
> > riffle [1,2,3] [4,5,6]
> [1,4,2,5,3,6]
> ```
>
> **Hint.** Use a list comprehension together with the library function `zip :: [a] -> [b] -> [(a,b)]` to combine the two lists.

Solution:

```
module Solution where
```

```haskell
import Library

riffle xs ys = concat [[x, y] | (x, y) <- zip xs ys]
```

Spec test:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

test_riffle_type :: [a] -> [a] -> [a]
test_riffle_type = riffle

prop_riffle_correct :: [Int] -> Property
prop_riffle_correct xs = forAll (vector $ length xs) $ \ys ->
  riffle xs ys === concat [[x, y] | (x, y) <- zip xs ys]
```

Histogram

> Write a function `histogram :: [Int] -> String` that takes a list of integers between 0 and 9 and outputs a vertical histogram showing the frequency of each number in the list. For example,
>
> ```
> > putStr (histogram [1,1,1,5])
>  *
>  *
>  *   *
> ==========
> 0123456789
>
> > putStr (histogram [1,3,4,3,6,6,3,4,2,4,9])
>    **
>    ** *
>  **** *  *
> ==========
> 0123456789
> ```
>
> Note that you must use `putStr` to actually visualize the histogram if you are testing your code in `ghci`, otherwise you get a textual representation of the string such as `"* *\n==========\n0123456789\n"`. Here on Weblab, the use of `putStr` is not required.
>
> **Hint.** You can use the function `unlines :: [String] -> String` to join a list of lines into a single string with newline characters in between.

Solution:

```
module Solution where

import Library

histogram :: [Int] -> String
histogram xs = unlines $
  [ [ if freqs !! i >= f then '*' else ' ' | i <- [0..9] ] | f <- [max_freq,max_frec
  ++
  [ "=========="
  , "0123456789"
  ]
  where
    freqs = [ length $ filter (==i) xs | i <- [0..9] ]
    max_freq = maximum freqs
```

Spec test:

```
module Test where

import Test.QuickCheck
import Library
import Solution

histogram_spec xs = unlines $
  [ [ if freqs !! i >= f then '*' else ' ' | i <- [0..9] ] | f <- [max_freq,max_frec
  ++
  [ "=========="
  , "0123456789"
  ]
  where
    freqs = [ length $ filter (==i) xs | i <- [0..9] ]
    max_freq = maximum freqs

prop_histogram_correct :: Property
prop_histogram_correct =
  forAll (listOf (chooseInt (0,9))) $ \xs ->
  histogram xs === histogram_spec xs
```

Local extrema

> Given a list of values of some type `a` that implements the `Ord` type class, the *local extrema* are the values that are either strictly bigger or strictly smaller than the numbers immediately before or after them. The goal of this question is to implement two different versions of the function `localExtrema :: Ord a => [a] -> [a]` that returns the list of all local extrema in a given list. The first and last elements of a list are never considered to be local extrema.

> Examples:
>
> ```
> localExtrema [] = []
> localExtrema [0,1,0] = [1]
> localExtrema [1,0,1] = [0]
> localExtrema [1,5,2,6,3,7] = [5,2,6,3]
> localExtrema [1,2,3,4,5] = []
> localExtrema [1,2,3,3,3,2,1] = []
> ```
>
> **Hint.** You can make use of your implementation of the `triplets` function in one of the previous exercises.
>
> **Note.** This was a sub-question on the exam of 16/4/2021.

Solution:

```haskell
module Solution where

import Library

triplets :: [a] -> [(a,a,a)]
triplets xs = zip3 xs (drop 1 xs) (drop 2 xs)

localExtrema :: Ord a => [a] -> [a]
localExtrema xs = [ y | (x,y,z) <- triplets xs, (x < y && y > z) || (x > y && y < z)
```

Spec test:

```haskell
module Test where

import Test.QuickCheck
import Library
import Solution

prop_localExtrema_test1 :: Bool
prop_localExtrema_test1 = localExtrema [] == ([] :: [Int])

prop_localExtrema_test2 :: Bool
prop_localExtrema_test2 = localExtrema [0,1,0] == [1]

prop_localExtrema_test3 :: Bool
prop_localExtrema_test3 = localExtrema [1,0,1] == [0]

prop_localExtrema_test4 :: Bool
prop_localExtrema_test4 = localExtrema [1,5,2,6,3,7] == [5,2,6,3]

prop_localExtrema_test5 :: Bool
prop_localExtrema_test5 = localExtrema [1,2,3,4,5] == []
```

```
prop_localExtrema_test6 :: Bool
prop_localExtrema_test6 = localExtrema [1,2,3,3,3,2,1] == []

triplets_spec :: [a] -> [(a,a,a)]
triplets_spec xs = zip3 xs (drop 1 xs) (drop 2 xs)

localExtrema_spec :: Ord a => [a] -> [a]
localExtrema_spec xs = [ y | (x,y,z) <- triplets_spec xs, (x < y && y > z) || (x > y

prop_localExtrema_correct :: [Int] -> Property
prop_localExtrema_correct xs = localExtrema xs === localExtrema_spec xs
```

Week 1B: Defining and testing functions

Recursive functions

Gotta sum 'em all

> Define a recursive function `sumdown :: Int -> Int` that returns the sum of the non-negative integers from a given value down to zero. For example, `sumdown 3` should return the result `3+2+1+0 = 6`.

Solution:

```
sumdown 0 = 0
sumdown n = n + sumdown (n-1)
```

Spec test:

```
prop_sumdown_zero :: Property
prop_sumdown_zero = within 1000000 $ sumdown 0 === 0

prop_sumdown_suc :: NonNegative Int -> Property
prop_sumdown_suc (NonNegative n) = within 1000000 $ sumdown (n+1) === (n+1) + sumdow

prop_sumdown_correct :: NonNegative Int -> Property
prop_sumdown_correct (NonNegative n) = within 1000000 $ sumdown n == (n * (n+1)) `di
```

Euclidian Algorithm

> Define a recursive function `euclid :: Int -> Int -> Int` that implements *Euclidean algorithm* for calculating the greatest common divisor of two non-negative integers. It works

> the following way:
>
> - If the two numbers are equal, this number is the result.
> - Otherwise, the smaller number is subtracted from the larger, and the same process is repeated with the smaller and the new number.
>
> For example:
>
> ```
> > euclid 6 27
> 3
> ```

Solution:

```haskell
euclid m n
  | m == n    = m
  | m < n     = euclid m (n - m)
  | otherwise = euclid (m - n) n
```

Spec test:

```haskell
prop_is_gcd :: Positive Int -> Positive Int -> Property
prop_is_gcd (Positive m) (Positive n) = within 1000000 $ euclid m n === gcd m n
```

Efficient exponentiation

> You are given an inefficient implementation of the `power` function that raises a number to the given power. For this assignment, the goal is to implement a more efficient version of this function that runs in O(log(N)) instead of O(n). To do this, your implementation should make use of the fact, that, if k is an even number, we can calculate n^k as follows:
>
> n^k= (n^2) k/2 = ($n \cdot n$) k/2(k even)
>
> So, instead of recursively using the case for k–1 we use the (much smaller) case for k/2. If k is not even, we simply go down one step to arrive at an even k: nk=$n \cdot nk$–1(k odd)
>
> Modify the definition of `power` to make use of this more efficient process.
>
> **Hint.** Haskell has built-in functions `even` and `odd` to check whether a number is even or odd. To divide integer numbers, use the `div` function (and not the function `(/)`, which is used to divide floating point and rational numbers).

Solution:

```haskell
power :: Integer -> Integer -> Integer
power n 0              = 1
power n k | even k     = power (n*n) (k `div` 2)
          | otherwise = n * power n (k-1)
```

Spec test:

```haskell
prop_power_correct :: Integer -> NonNegative Integer -> Property
prop_power_correct n (NonNegative k) = within 1000000 $ power n k === n ^ k

prop_power_efficient :: Integer -> Property
prop_power_efficient n = within 100000 $ power n 100000 === n ^ 100000
```

Towers of hanoi

> The *Towers of Hanoi* (Wikipedia) is a classic puzzle with a solution that can be described recursively. Disks of different sizes are stacked on three pegs; the goal is to get from a starting configuration with all disks stacked on the first peg to an ending configuration with all disks stacked on the last peg. The rules are as follows:
>
> - Only one disk can be moved at a time
> - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
> - No larger disk may be placed on top of a smaller disk.
>
> The recursive solution to this problem can be solved as follows. If only one peg has to be moved, move it from the source peg to the target peg directly. If (m) pegs ((m > 0)) have to be moved, proceed as follows:
>
> - Move (m-1) disks from the source peg to the spare peg, by applying this procedure recursively.
> - Move the largest disk from the source peg to the target peg.
> - Move (m-1) disks from the spare peg to the target peg, again applying this procedure recursively.
>
> The goal of this exercise is to implement the function `hanoi :: Int -> Peg -> Peg -> Peg -> [Move]` , where `type Peg = String` and `type Move = (Peg,Peg)` are type synonyms, such that `hanoi n source target spare` computes the list of moves for solving the puzzle with `n` disks. For example,

> ```
> > hanoi 2 "a" "b" "c"
> [("a","c"),("a","b"),("c","b")]
> ```

Solution:

```
hanoi n source target spare
 | n == 0 = []
 | n > 0  = hanoi (n-1) source spare target ++ [(source,target)] ++ hanoi (n-1) spar
```

Spec test:

```
prop_hanoi_one :: String -> String -> String -> Property
prop_hanoi_one source target spare = within 100000 $ hanoi 1 source target spare ===

prop_hanoi_step :: String -> String -> String -> Property
prop_hanoi_step source target spare =
  forAll (chooseInt (1,10)) $ \n ->
  within 1000000 $
  hanoi n source target spare === hanoi (n-1) source spare target ++ [(source,target
```

Recursion on Lists

Element the third

> Define functions `third1 third2 third3 :: [a] -> a` that all return the third element in a list that contains at least this many elements.
>
> - `third1` should be defined in terms of `head` and `tail`
> - `third2` should be defined using `!!`
> - `third3` should be defined using pattern matching

Solution:

```
third1 xs = head (tail (tail xs))

third2 xs = xs !! 2

third3 (_:_:x:_) = x
```

Spec test:

```
prop_third1_correct :: Int -> Int -> Int -> [Int] -> Property
```

```
prop_third1_correct x y z xs = within 1000000 $ third1 (x:y:z:xs) === z

prop_third2_correct :: Int -> Int -> Int -> [Int] -> Property
prop_third2_correct x y z xs = within 1000000 $ third2 (x:y:z:xs) === z

prop_third3_correct :: Int -> Int -> Int -> [Int] -> Property
prop_third3_correct x y z xs = within 1000000 $ third3 (x:y:z:xs) === z
```

Product

> Implement a function `product` that produces the product of a list of numbers. For example, `product [2,3,4] = 24`.

Solution:

```
import Prelude hiding (product)

product [] = 1
product (x:xs) = x * (product xs)
```

Spec test:

```
import Prelude hiding (product)

prop_product_empty :: Property
prop_product_empty = within 1000000 $ product [] === 1

prop_product_cons :: Int -> [Int] -> Property
prop_product_cons x xs = within 1000000 $ product (x:xs) === x * product xs
```

Reverse that list!

> Implement the function `reverse` that reverses the elements of a list.

Solution:

```
import Prelude hiding (reverse)

reverse_helper :: [a] -> [a] -> [a]
reverse_helper [] ys = ys
reverse_helper (x:xs) ys = reverse_helper xs (x:ys)

reverse xs = reverse_helper xs []
```

Spec test:

```haskell
import Prelude hiding (reverse)
import qualified Prelude

prop_reverse_nil :: Property
prop_reverse_nil = within 1000000 $ reverse ([] :: [Int]) === []

prop_reverse_cons :: Int -> [Int] -> Property
prop_reverse_cons x xs = within 1000000 $ reverse (x:xs) === reverse xs ++ [x]

prop_reverse_correct :: [Int] -> Property
prop_reverse_correct xs = within 1000000 $ reverse xs === Prelude.reverse xs
```

Standard functions on lists

> Redefine the following functions from the Prelude using recursion:
>
> - The function `and :: [Bool] -> Bool` deciding if all logical values in a list are `True`
> - The function `concat :: [[a]] -> [a]` concatenating a list of lists.
> - The function `replicate :: Int -> a -> [a]` producing a list with `n` identical elements
> - The function `(!!) :: [a] -> Int -> a` selecting the `n` th element of a list
> - The function `elem :: Eq a => a -> [a] -> Bool` deciding if a value is an element of the list.
> - The function `sum :: [Int] -> Int` calculating the sum of a list of numbers.
> - The function `take :: Int -> [a] -> [a]` taking a given number of elements from the start of a list.
> - The function `last :: [a] -> a` selecting the last element of a non-empty list.

Solution:

```haskell
import Prelude hiding (and, concat, replicate, (!!), elem, sum, take, last)

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

concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss

replicate :: Int -> a -> [a]
replicate 0 x = []
replicate n x
  | n > 0     = x : replicate (n-1) x
```

```haskell
  | otherwise = undefined

(!!) :: [a] -> Int -> a
(x:xs) !! 0   = x
(x:xs) !! n
  | n > 0     = xs !! (n-1)
  | otherwise = undefined

elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys)
  | x == y    = True
  | otherwise = elem x ys

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

take :: Int -> [a] -> [a]
take 0 xs     = []
take n []     = []
take n (x:xs)
  | n > 0     = x : (take (n-1) xs)
  | otherwise = undefined

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

Spec test:

```haskell
import Prelude hiding (and, concat, replicate, (!!), elem, sum, take, last)
import qualified Prelude

prop_and_empty :: Property
prop_and_empty = within 1000000 $ and [] === True

prop_and_cons :: Bool -> [Bool] -> Property
prop_and_cons b bs = within 1000000 $ and (b:bs) === (b && and bs)

prop_concat_empty :: Property
prop_concat_empty = within 1000000 $ concat ([] :: [[Int]]) === []

prop_concat_cons :: [Int] -> [[Int]] -> Property
prop_concat_cons xs xss = within 1000000 $ concat (xs:xss) === (xs ++ concat xss)

prop_replicate_zero :: Int -> Property
prop_replicate_zero x = within 1000000 $ replicate 0 x === []

prop_replicate_suc :: Int -> Property
```

```haskell
prop_replicate_suc x = forAll (chooseInt (0,10)) $ \n -> within 1000000 $
  replicate (n+1) x === x : replicate n x

prop_select_head :: Int -> [Int] -> Property
prop_select_head x xs = within 1000000 $ (x:xs) !! 0 === x

prop_select_int :: Property
prop_select_int = forAll (chooseInt (0,10)) $ \n -> within 1000000 $
  [0..] !! n === n

prop_elem_empty :: Int -> Property
prop_elem_empty x = within 1000000 $ not (x `elem` [])

prop_elem_single :: Int -> Property
prop_elem_single x = within 1000000 $ x `elem` [x]

prop_elem_later :: Property
prop_elem_later = forAll (chooseInt (0,10)) $ \n -> within 1000000 $
  True `elem` (Prelude.replicate n False ++ [True])

prop_sum_empty :: Property
prop_sum_empty = within 1000000 $ sum [] === 0

prop_sum_cons :: Int -> [Int] -> Property
prop_sum_cons x xs = within 1000000 $ sum (x:xs) === x + sum xs

prop_take_zero :: [Int] -> Property
prop_take_zero xs = within 1000000 $ take 0 xs === []

prop_take_n :: Property
prop_take_n = forAll (chooseInt (0,10)) $ \n -> within 1000000 $
  take n [1..] === [1..n]

prop_last_single :: Int -> Property
prop_last_single x = within 1000000 $ last [x] === x

prop_last_n :: Property
prop_last_n = forAll (chooseInt (0,10)) $ \n -> within 1000000 $
  last [0..n] === n
```

Merge sort:

> Define a recursive function `merge :: Ord a => [a] -> [a] -> [a]` that merges two sorted lists to give a single sorted list. For example:
>
> ```
> > merge [2,5,6] [1,3,4]
> [1,2,3,4,5,6]
> ```
>
> Note: your definition should not use other functions on sorted lists such as `insert` or

> `isort`, but should be defined using explicit recursion.
>
> Next, define a function `split :: [a] -> ([a],[a])` that splits a list into two halves whose lengths differ by at most one.
>
> Using `merge` and `split`, define a function `msort :: Ord a => [a] -> [a]` that implements *merge sort*, in which the empty list and singleton lists are already sorted, and any other list is sorted by merging together the two lists that result from sorting the two halves of the list separately.

Solution:

```haskell
merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys)
  | x < y           = x : merge xs (y:ys)
  | otherwise       = y : merge (x:xs) ys

split xs = (take k xs, drop k xs)
  where k = length xs `div` 2

msort []  = []
msort [x] = [x]
msort xs  = merge (msort ys) (msort zs)
  where (ys, zs) = split xs
```

Spec test:

```haskell
import qualified Data.List as List

prop_merge_length :: SortedList Int -> SortedList Int -> Property
prop_merge_length (Sorted xs) (Sorted ys) = within 1000000 $ length (merge xs ys) ==

prop_merge_sorted :: SortedList Int -> SortedList Int -> Property
prop_merge_sorted (Sorted xs) (Sorted ys) = within 1000000 $ is_sorted (merge xs ys)
  where
    is_sorted xs = List.sort xs === xs

prop_split_same_length :: [Int] -> Property
prop_split_same_length xs = within 1000000 $ abs (length xs1 - length xs2) <= 1
  where
    (xs1,xs2) = split xs

prop_split_join :: [Int] -> Property
prop_split_join xs = within 1000000 $ xs === xs1 ++ xs2
  where
    (xs1,xs2) = split xs
```

```
prop_msort_correct :: [Int] -> Property
prop_msort_correct xs = within 1000000 $ msort xs === List.sort xs
```

Bag equality

> Two lists are 'bag equal' if they contain the same elements, but possibly in a different order. Implement a function `bagEqual :: (Eq a) => [a] -> [a] -> Bool` that checks if two given lists are bag equal.
>
> Hint: you can make use of the library functions `elem :: (Eq a) => a -> [a] -> Bool` and `delete :: (Eq a) => a -> [a] -> [a]`.

Solution:

```
import Data.List (elem, delete)

bagEqual [] [] = True
bagEqual (x:xs) [] = False
bagEqual [] (y:ys) = False
bagEqual (x:xs) ys
    | elem x ys = bagEqual xs (delete x ys)
    | otherwise = False
```

Spec test:

```
import Data.List
import qualified Data.Map as Map

prop_bagEqual_perm :: [Int] -> Property
prop_bagEqual_perm xs = forAll (shuffle xs) $ \ys -> bagEqual xs ys

prop_bagEqual_oneOff :: Int -> [Int] -> Property
prop_bagEqual_oneOff x xs = forAll (shuffle (x:xs)) $ \ys -> forAll (shuffle ((x+1):

freqs_spec :: (Ord a) => [a] -> Map.Map a Int
freqs_spec = foldr (\x -> Map.insertWith (+) x 1) Map.empty

prop_bagEqual_correct :: [Int] -> [Int] -> Property
prop_bagEqual_correct xs ys = bagEqual xs ys === (freqs_spec xs == freqs_spec ys)
```

Bank card numbers

> Define a function `luhn :: [Int] -> Bool` that implements the *Luhn algorithm* to check if a given bank card number is valid.
>
> As a reminder, the *Luhn algorithm* (Wikipedia) is used to check bank card numbers for simple

> errors such as mistyping a digit, and proceeds as follows:
>
> - consider each digit as a separate number;
> - moving left, double every other number from the second last;
> - subtract 9 from each number that is now greater than 9;
> - add all the resulting numbers together;
> - if the total is divisible by 10, the card number is valid.

Solution:

```
luhnDouble x = if double_x > 9 then double_x - 9 else double_x
  where double_x = 2*x

luhnSum [] = 0
luhnSum [x] = x
luhnSum (x1:x2:xs) = x1 + luhnDouble x2 + luhnSum xs

luhn xs = luhnSum (reverse xs) `mod` 10 == 0
```

Spec test:

```
digit :: Gen Int
digit = choose (0, 9)

digits :: Gen [Int]
digits = listOf1 digit

prop_luhn_single_digit :: Property
prop_luhn_single_digit = forAll digit $ \x -> luhn [x] == (x `mod` 10 == 0)

prop_luhn_double_digit_small :: Property
prop_luhn_double_digit_small =
  forAll (choose (1,4) :: Gen Int) $ \x -> luhn [x, 10-2*x]

prop_luhn_double_digit_large :: Property
prop_luhn_double_digit_large = forAll (choose (5,9) :: Gen Int) $ \x -> luhn [x, 19-

luhn_spec xs = luhnSum (reverse xs) `mod` 10 == 0
  where
    luhnDouble x = if double_x > 9 then double_x - 9 else double_x
      where double_x = 2*x

    luhnSum [] = 0
    luhnSum [x] = x
    luhnSum (x1:x2:xs) = x1 + luhnDouble x2 + luhnSum xs
```

```haskell
prop_luhn_correct :: Property
prop_luhn_correct = forAll digits (\xs -> luhn xs === luhn_spec xs)
```

Local maxima

> A *local maximum* of a list is an element of the list that is strictly greater than the elements right before and after it. For example, in the list `[3,5,2,3,4]`, the only local maximum is `5`, since it is both greater than `3` and greater than `2`. `4` is not a local maximum because there is no element that comes after it.
>
> Write a function `localMaxima :: [Int] -> [Int]` that computes all the local maxima in the given list and returns them in order. For example
>
> ```
> > localMaxima [2,9,5,6,1]
> [9,6]
> > localMaxima [2,3,4,1,5]
> [4]
> > localMaxima [1,2,3,4,5]
> []
> ```

Solution:

```haskell
localMaxima :: [Int] -> [Int]
localMaxima []                         = []
localMaxima (x:y:z:xs) | x<y && y>z = y:localMaxima (y:z:xs)
localMaxima (x:xs)                     = localMaxima xs
```

Spec test:

```haskell
prop_single_local_maximum :: Int -> Int -> Int -> Property
prop_single_local_maximum x y z = within 1000000 $ localMaxima [x,y',z] === [y']
  where y' = 1 + abs x + abs y + abs z

prop_no_local_maxima :: NonNegative Int -> Property
prop_no_local_maxima (NonNegative n) = within 1000000 $
      localMaxima [0..n] === []
  .&. localMaxima [n,(n-1)..0] === []

localMaxima_spec :: [Int] -> [Int]
localMaxima_spec []                         = []
localMaxima_spec (x:y:z:xs) | x<y && y>z = y:localMaxima_spec (y:z:xs)
localMaxima_spec (x:xs)                     = localMaxima_spec xs

prop_localMaxima_correct :: [Int] -> Property
prop_localMaxima_correct xs = within 1000000 $ localMaxima xs === localMaxima_spec x
```

Testing functions with QuickCheck

Testing the sum

> The standard Haskell function `sum :: Num a => [a] -> a` is fully defined by the following properties:
> - `sum [] = 0`
> - `sum [x] = x`
> - `sum (xs ++ ys) = sum xs + sum ys`
> Write three property tests `prop_sum_empty` , `prop_sum_singleton` , and `prop_sum_concat` to verify that these properties indeed hold.

Solution:

```
prop_sum_empty = sum [] == 0

prop_sum_singleton x = sum [x] == x

prop_sum_concat xs ys = sum (xs ++ ys) == sum xs + sum ys
```

Spec tests:

```
prop_sum_empty' :: Bool
prop_sum_empty' = prop_sum_empty

prop_sum_singleton' :: Int -> Bool
prop_sum_singleton' = prop_sum_singleton

prop_sum_concat' :: [Int] -> [Int] -> Bool
prop_sum_concat' = prop_sum_concat
```

Testing the sort

> Suppose we have a function `sort :: Ord a => [a] -> [a]` . In order to test this function, we can write a property that the output is always sorted:
>
> ```
> sorted :: Ord a => [a] -> Bool
> sorted (x:y:ys) = x <= y && sorted (y:ys)
> sorted _        = True
>
> prop_sort_sorted :: [Int] -> Bool
> prop_sort_sorted xs = sorted (sort xs)
> ```
>
> However, this is not enough to fully specify the sort function: a trivial definition such as `sort`

> `xs = []` also satisfies it. We also need to test that the input and output have the same elements.
>
> Implement a function `sameElements :: Eq a => [a] -> [a] -> Bool` that returns `True` if the two given lists have precisely the same elements (but possibly in a different order). Then write a test `prop_sort_sameElements` to test that the input and output of the `sort` function always have the same elements.

Solution:

```
import Data.List (sort, delete)

sorted :: Ord a => [a] -> Bool
sorted (x:y:ys) = x <= y && sorted (y:ys)
sorted _        = True

prop_sort_sorted :: [Int] -> Bool
prop_sort_sorted xs = sorted (sort xs)

sameElements :: Eq a => [a] -> [a] -> Bool
sameElements [] [] = True
sameElements [] (y:ys) = False
sameElements (x:xs) ys
  | x `elem` ys = sameElements xs (delete x ys)
  | otherwise   = False

prop_sort_sameElements :: [Int] -> Bool
prop_sort_sameElements xs = sameElements xs (sort xs)
```

Spec test:

```
prop_sameElements_same :: [Int] -> Bool
prop_sameElements_same xs = sameElements xs xs

prop_sameElements_sym :: [Int] -> [Int] -> Property
prop_sameElements_sym xs ys = sameElements xs ys === sameElements ys xs

prop_sameElements_shuffle :: [Int] -> Property
prop_sameElements_shuffle xs = forAll (shuffle xs) (\ys -> sameElements xs ys)

prop_sameElements_length :: Int -> [Int] -> Property
prop_sameElements_length x xs = forAll (shuffle $ x:xs) (\ys -> not (sameElements xs

prop_sameElements_oneOff :: Int -> [Int] -> Property
prop_sameElements_oneOff x xs = forAll (shuffle $ (x+1):xs) (\ys -> not (sameElement

prop_sort_sameElements' :: [Int] -> Bool
prop_sort_sameElements' xs = prop_sort_sameElements xs
```

Testing the index

> There are several ways to get the `n` th element of a list in Haskell. There is of course the builtin function `(!!)` , but we can for example also use `drop` to remove the first `n` elements and then take the `head` of the list. We can try to test that the two methods are equivalent as follows:
>
> ```
> prop_index :: [Int] -> Int -> Bool
> prop_index xs n = xs !! n == head (drop n xs)
> ```
>
> However, this results in an error:
>
> ```
> Property prop_index failed!
> *** Failed! Exception: 'Prelude.!!: index too large' (after 1 test):
> []
> 0
> ```
>
> Fix the test so that it tests the correct property.
>
> **Hint.** You'll probably need to change the return type of `prop_index` from `Bool` to `Property` .

Solution:

```
import Test.QuickCheck

prop_index :: [Int] -> Int -> Property
prop_index xs n = (n >= 0 && n < length xs) ==> xs !! n == head (drop n xs)
```

Spec test:

```
prop_index' :: [Int] -> Int -> Property
prop_index' = prop_index
```

Testing the halve

> In one of the first exercises, you implemented a function `halve :: [a] -> ([a],[a])` that splits an even-lengthed list into two halves. Now write a QuickCheck property `prop_halve_sameLength` to test the following property: if the input is a list of even length, then the two halves have the same length.

Solution:

```haskell
import Test.QuickCheck

prop_halve_sameLength :: [Int] -> Property
prop_halve_sameLength xs = length xs `mod` 2 == 0 ==> sameLength (halve xs)
  where
    sameLength (ys,zs) = length ys == length zs
```

Spec test:

```haskell
prop_halve_sameLength' :: [Int] -> Property
prop_halve_sameLength' = prop_halve_sameLength
```

Testing all the functions

> In the 'Library' tab, there are four functions defined that work on sorted lists: `elemSorted` , `insertSorted` , `deleteSorted` , and `mergeSorted` .
>
> - `elemSorted` checks if the given value is an element of the list. It assumes that the input list is sorted.
> - `insertSorted` inserts an element into a sorted list. If the element is already present in a list, it returns the list unchanged.
> - `deleteSorted` removes an element from a list. It assumes that the input list is sorted and has no duplicates.
> - `mergeSort` merges two sorted lists into a single sorted list. It assumes that the input lists have no duplicates, and ensures that the output list also has no duplicates.
>
> This time their implementation is actually correct! Verify this by copying all the tests you wrote for the previous four assignments to the Solution tab here.

Solution:

```haskell
import Test.QuickCheck
import Data.List (sort)

prop_insertSorted :: Int -> Int -> Bool
prop_insertSorted x y = elemSorted y (insertSorted x [y])

prop_deleteSorted :: Int -> Int -> Property
prop_deleteSorted x y = x /= y ==> forAll genSortedList (\xs -> elemSorted x (delete

prop_insertDeleteSorted :: Int -> Property
prop_insertDeleteSorted x = forAll genSortedList (\xs -> not (elemSorted x (deleteSo

prop_mergeSorted :: Property
```

```
prop_mergeSorted = forAll genSortedList (\xs -> forAll genSortedList (\ys -> isSorte
  where
    isSorted :: [Int] -> Bool
    isSorted xs = sort xs == xs
```

Spec test:

```
prop_insertSorted' = prop_insertSorted

prop_deleteSorted' = prop_deleteSorted

prop_insertDeleteSorted' = prop_insertDeleteSorted

prop_mergeSorted' = prop_mergeSorted
```

Week 2A: Data Types

Parents and children

> **Step 1.** Define a data type `Person` with the following two constructors:
>
> - A constructor `Adult` with 4 fields: a first name of type `String` , a last name of type `String` , an age of type `Int` , and a job of type `Occupation` .
> - A constructor `Child` with 3 fields: a first name of type `String` , an age of type `Int` , and a grade level of type `Int` .
>
> The type `Occupation` should itself also be a data type with at least two constructors `Engineer` and `Lawyer` (both with no arguments), plus any other cases you come up with.
>
> You can take a look at the `Test` tab for some examples that should compile with your definitions.
>
> **Step 2.** Implement a function `giveFullName :: Person -> String` that for an adult returns their first and last name with a space in between, and for a child just their first name.

Solution:

```
data Person = Adult String String Int Occupation | Child String Int Int

data Occupation = Lawyer | Engineer

giveFullName :: Person -> String
giveFullName (Adult first last _ _) = first ++ " " ++ last
giveFullName (Child first _ _) = first
```

Spec test:

```
import Control.DeepSeq

instance NFData Occupation where
  rnf Lawyer   = ()
  rnf Engineer = ()

instance Arbitrary Occupation where
  arbitrary = elements [Lawyer, Engineer]

instance NFData Person where
  rnf (Adult x y z w) = rnf (x,y,z,w)
  rnf (Child x y z) = rnf (x,y,z)

prop_occupations_total :: Blind Occupation -> Property
prop_occupations_total (Blind x) = within 1000000 $ total x

prop_adult_total :: String -> String -> Int -> Blind Occupation -> Property
prop_adult_total x y z (Blind w) = within 1000000 $ total (Adult x y z w)

prop_child_total :: String -> Int -> Int -> Property
prop_child_total x y z = within 1000000 $ total (Child x y z)

prop_fullName_adult_total :: String -> String -> Int -> Blind Occupation -> Property
prop_fullName_adult_total x y z (Blind w) = within 1000000 $ giveFullName (Adult x y

prop_fullName_child :: String -> Int -> Int -> Property
prop_fullName_child x y z = within 1000000 $ giveFullName (Child x y z) === x
```

Type Synonyms

> Type synonyms are often useful to draw a semantic difference between two items that have the same type but a different meaning, which can help the user of the function to avoid making mistakes. For example, instead of a function `login :: String -> String -> LoginResult` we can define `type Username = String` and `type Password = String` and then give a more informative type `login :: Username -> Password -> LoginResult` .

> In the Test tab there are some examples of functions that do this, but the type synonyms are missing. Write down type synonyms to make the examples compile!

Solution:

```haskell
type Principal = Double
type Rate = Double

type Slope = Double
type Intercept = Double
type XCoordinate = Double
type YCoordinate = Double

type Name = String
type Occupation = String
```

Spec test:

```haskell
calculateMonthlyInterest :: Principal -> Rate -> Double
calculateMonthlyInterest p r = (p * r) / 12.0

calculateY :: Slope -> Intercept -> XCoordinate -> YCoordinate
calculateY slope intercept x = slope * x + intercept

greet :: Name -> Occupation -> String
greet n o = "Hello, my name is " ++ n ++ ". I am a " ++ o ++ "."

prop_ok = True
```

Binary search trees

> Consider the following type of binary trees:
>
> ```haskell
> data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)
> ```
>
> A binary tree is a *search tree* if for every node, all values in the left subtree are smaller than the stored value, and all values in the right subtree are greater than the stored value.
>
> A tree is *balanced* if the number of leaves in the left and right subtree of every node differs by at most one.
>
> **Assignment 1.** Define a function `occurs :: Ord a => a -> Tree a -> Bool` that checks if a value occurs in the given search tree. **Hint:** the standard prelude defines a type `data`

> `Ordering = LT | EQ | GT` together with a function `compare :: Ord a => a -> a -> Ordering` that decides if one value in an ordered type is less than (`LT`), equal to (`EQ`), or greater than (`GT`) another value.
>
> **Assignment 2.** Define a function `is_balanced :: Tree a -> Bool` that checks if the given tree is balanced. Hint: first define a function that returns the number of elements in a tree.
>
> **Assignment 3.** Define a function `flatten :: Tree a -> [a]` that returns a list that contains all the elements stored in the given tree from left to right.
>
> **Assignment 4.** Define a function `balance :: [a] -> Tree a` that converts a non-empty list into a balanced (not necessarily search) tree.
>
> The functions you define should satisfy `flatten (balance xs) == xs` for any list `xs` .

Solution:

```
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)
  deriving (Show, Eq)

occurs x Empty        = False
occurs x (Leaf y)     = x == y
occurs x (Node l y r) = case compare x y of
  LT -> occurs x l
  EQ -> x == y
  GT -> occurs x r

count_elements :: Tree a -> Int
count_elements Empty = 0
count_elements (Leaf x) = 1
count_elements (Node l x r) = 1 + count_elements l + count_elements r

is_balanced :: Tree a -> Bool
is_balanced Empty = True
is_balanced (Leaf x) = True
is_balanced (Node l x r) =
     abs (count_elements l - count_elements r) <= 1
  && is_balanced l
  && is_balanced r

flatten :: Tree a -> [a]
flatten Empty = []
flatten (Leaf x) = [x]
flatten (Node l x r) = flatten l ++ [x] ++ flatten r

balance :: [a] -> Tree a
balance []  = Empty
balance [x] = Leaf x
balance xs  = Node (balance ys) x (balance zs)
  where
```

```
    n      = length xs `div` 2
    ys     = take n xs
    (x:zs) = drop n xs
```

Spec test:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized tree'
    where
      tree' 0       = oneof [pure Empty, Leaf <$> arbitrary]
      tree' n | n>0 = oneof [pure Empty, Leaf <$> arbitrary, Node <$> tree' m <*> ar
        where m = n `div` 2

  shrink Empty        = []
  shrink (Leaf x)     = []
  shrink (Node l x r) = [l,r]

count_elements_spec :: Tree a -> Int
count_elements_spec Empty = 0
count_elements_spec (Leaf x) = 1
count_elements_spec (Node l x r) = 1 + count_elements_spec l + count_elements_spec r

prop_occurs_empty :: Int -> Property
prop_occurs_empty x = within 1000000 $ not (occurs x Empty)

prop_occurs_leaf :: Int -> Property
prop_occurs_leaf x = within 1000000 $
  occurs x (Leaf x) .&. not (occurs x (Leaf (x+1)))

balance_spec :: [a] -> Tree a
balance_spec []  = Empty
balance_spec [x] = Leaf x
balance_spec xs  = Node (balance_spec ys) x (balance_spec zs)
  where
    n      = length xs `div` 2
    ys     = take n xs
    (x:zs) = drop n xs

is_balanced_spec :: Tree a -> Bool
is_balanced_spec Empty = True
is_balanced_spec (Leaf x) = True
is_balanced_spec (Node l x r) =
     abs (count_elements_spec l - count_elements_spec r) <= 1
  && is_balanced_spec l
  && is_balanced_spec r

prop_occurs_bool :: Bool -> SortedList Bool -> Property
prop_occurs_bool x (Sorted xs) = within 1000000 $ occurs x (balance_spec xs) === (x

prop_occurs_int :: Int -> SortedList Int -> Property
```

```haskell
prop_occurs_int x (Sorted xs) = within 1000000 $ occurs x (balance_spec xs) === (x `

prop_flatten_correct :: NonEmptyList Int -> Property
prop_flatten_correct (NonEmpty xs) = within 1000000 $
  flatten (balance_spec xs) === xs

prop_balance_idempotent :: NonEmptyList Int -> Property
prop_balance_idempotent (NonEmpty xs) = within 1000000 $
  balance (flatten (balance xs)) === balance xs

prop_balance_is_balanced :: NonEmptyList Int -> Property
prop_balance_is_balanced (NonEmpty xs) = within 1000000 $
  is_balanced_spec (balance xs)

prop_flatten_balance :: NonEmptyList Int -> Property
prop_flatten_balance (NonEmpty xs) = within 1000000 $
  flatten (balance xs) === xs

prop_balance_bool :: Bool -> SortedList Bool -> Property
prop_balance_bool x (Sorted xs) = within 1000000 $ occurs x (balance xs) === (x `ele

prop_balance_int :: Int -> SortedList Int -> Property
prop_balance_int x (Sorted xs) = within 1000000 $ occurs x (balance xs) === (x `elem
```

Arithmetic expressions

> Consider the type of arithmetic expressions involving `+` and `-` :
>
> ```haskell
> data Expr = Val Int | Add Expr Expr | Subs Expr Expr
> ```
>
> 1. Define a function `size :: Expr -> Int` that calculates the number of values in an expression.
> 2. Define a function `eval :: Expr -> Int` that evaluates an expression to an integer value.

Solution:

```haskell
data Expr = Val Int | Add Expr Expr | Subs Expr Expr
  deriving (Show, Eq)

eval :: Expr -> Int
eval (Val x) = x
eval (Add e1 e2) = eval e1 + eval e2
eval (Subs e1 e2) = eval e1 - eval e2

size :: Expr -> Int
size (Val _) = 1
```

```haskell
size (Add e1 e2) = size e1 + size e2
size (Subs e1 e2) = size e1 + size e2
```

Spec test:

```haskell
instance Arbitrary Expr where
  arbitrary = sized expr'
    where
      expr' 0       = Val <$> arbitrary
      expr' n | n>0 = oneof [Val <$> arbitrary, Add <$> expr' m <*> expr' m, Subs <$
        where m = n `div` 2

  shrink (Val _)      = []
  shrink (Add e1 e2)  = [e1,e2]
  shrink (Subs e1 e2) = [e1,e2]

folde :: (Int -> a) -> (a -> a -> a) -> (a -> a -> a) -> Expr -> a
folde f g h (Val x) = f x
folde f g h (Add e1 e2) = g (folde f g h e1) (folde f g h e2)
folde f g h (Subs e1 e2) = h (folde f g h e1) (folde f g h e2)

prop_eval_val :: Int -> Property
prop_eval_val x = within 1000000 $ eval (Val x) === x

prop_eval_plus :: Int -> Int -> Property
prop_eval_plus x y = within 1000000 $ eval (Add (Val x) (Val y)) === x + y

prop_eval_minus :: Int -> Int -> Property
prop_eval_minus x y = within 1000000 $ eval (Subs (Val x) (Val y)) === x - y

prop_eval_correct :: Expr -> Property
prop_eval_correct e = within 1000000 $ eval e === folde id (+) (-) e

prop_size_val :: Int -> Property
prop_size_val x = within 1000000 $ size (Val x) === 1

prop_size_plus :: Int -> Int -> Property
prop_size_plus x y = within 1000000 $ size (Add (Val x) (Val y)) === 2

prop_size_minus :: Int -> Int -> Property
prop_size_minus x y = within 1000000 $ size (Subs (Val x) (Val y)) === 2

prop_size_correct :: Expr -> Property
prop_size_correct e = within 1000000 $ size e === folde (const 1) (+) (+) e
```

Tautology Checker

> You are given a tautology checker for boolean propositions (see Section 8.6 of the book).

> **Assignment 1.** Extend the tautology checker to support the use of logical disjunction (`\/`) and equivalence (`<=>`) of propositions. The new constructors should be called `Or` and `Equiv` (otherwise the tests will not work).
>
> **Assignment 2.** Implement a function `isSat :: Prop -> Maybe Subst` that returns `Just s` if there is a substitution `s` for which the given proposition is true, and `Nothing` if there is no such substitution.

Solution:

```
import Data.List (nub) -- The function 'nub' removes duplicates from a list

data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Or Prop Prop
          | Imply Prop Prop
          | Equiv Prop Prop
  deriving (Show)

type Assoc k v = [(k,v)]

find :: (Eq k) => k -> Assoc k v -> v
find k [] = error "Key not found!"
find k ((k',x):xs)
  | k == k'   = x
  | otherwise = find k xs

type Subst = Assoc Char Bool


eval :: Subst -> Prop -> Bool
eval _ (Const b)   = b
eval s (Var x)     = find x s
eval s (Not p)     = not (eval s p)
eval s (And p q)   = eval s p && eval s q
eval s (Or p q)    = eval s p || eval s q
eval s (Imply p q) = eval s p <= eval s q
eval s (Equiv p q) = eval s p == eval s q

vars :: Prop -> [Char]
vars (Const _)   = []
vars (Var x)     = [x]
vars (Not p)     = vars p
vars (And p q)   = vars p ++ vars q
vars (Or p q)    = vars p ++ vars q
vars (Imply p q) = vars p ++ vars q
vars (Equiv p q) = vars p ++ vars q
```

```haskell
bools :: Int -> [[Bool]]
bools 0       = [[]]
bools n | n>0 = map (False:) bss ++ map (True:) bss
  where bss = bools (n-1)

substs :: Prop -> [Subst]
substs p = map (zip vs) (bools (length vs))
  where vs = nub (vars p)

isTaut :: Prop -> Bool
isTaut p = and [eval s p | s <- substs p]

isSat :: Prop -> Maybe Subst
isSat p = if null sats then Nothing else Just (head sats)
  where
    sats = [ s | s <- substs p , eval s p ]


-- Checking whether a proposition is satisfiable is known as the Boolean
-- Satisfiability Problem (https://en.wikipedia.org/wiki/Boolean_satisfiability_prob
-- which is a famous NP-complete problem. Hence it is not possible to
-- give a polynomial implementation (unless P = NP).
```

Spec tests:

```haskell
instance Arbitrary Prop where
  arbitrary = sized expr'
    where
      expr' 0       = oneof [Const <$> arbitrary, Var <$> elements "pqrst"]
      expr' n | n>0 = oneof
        [ Const <$> arbitrary
        , Var   <$> elements "pqrst"
        , Not   <$> expr' (n-1)
        , And   <$> expr' (n `div` 2) <*> expr' (n `div` 2)
        , Or    <$> expr' (n `div` 2) <*> expr' (n `div` 2)
        , Imply <$> expr' (n `div` 2) <*> expr' (n `div` 2)
        , Equiv <$> expr' (n `div` 2) <*> expr' (n `div` 2)
        ]

  shrink (Const _)     = []
  shrink (Var x)       = [Const True, Const False        ] ++ [ Var x'        | x' <
  shrink (Not e)       = [Const True, Const False, e     ] ++ [ Not e'        | e'
  shrink (And e1 e2)   = [Const True, Const False, e1, e2] ++ [ And e1' e2'   | (e1'
  shrink (Or  e1 e2)   = [Const True, Const False, e1, e2] ++ [ Or  e1' e2'   | (e1'
  shrink (Imply e1 e2) = [Const True, Const False, e1, e2] ++ [ Imply e1' e2' | (e1'
  shrink (Equiv e1 e2) = [Const True, Const False, e1, e2] ++ [ Equiv e1' e2' | (e1'

genSubst :: Gen Subst
genSubst = (\(b1,b2,b3,b4,b5) -> zip "pqrst" [b1,b2,b3,b4,b5]) <$> arbitrary
```

```haskell
prop_eval_or :: Prop -> Prop -> Property
prop_eval_or e1 e2 = forAll genSubst $ \s -> eval s (Or e1 e2) === (eval s e1 || eva

prop_eval_equiv :: Prop -> Prop -> Property
prop_eval_equiv e1 e2 = forAll genSubst $ \s -> eval s (Equiv e1 e2) === (eval s e1

prop_vars_or :: Prop -> Prop -> Property
prop_vars_or e1 e2 = vars (Or e1 e2) === vars e1 ++ vars e2

prop_vars_equiv :: Prop -> Prop -> Property
prop_vars_equiv e1 e2 = vars (Equiv e1 e2) === vars e1 ++ vars e2

prop_isTaut_complete :: Prop -> Property
prop_isTaut_complete e =
  forAll genSubst $ \s ->
    isTaut e ==> eval s e == True

prop_isTaut_sound :: Prop -> Property
prop_isTaut_sound e =
  forAll genSubst $ \s ->
    eval s e == False ==> not (isTaut e)

prop_isSat_sound :: Prop -> Property
prop_isSat_sound e = maybe (property True) (\s -> eval s e === True) (isSat e)

prop_isSat_complete :: Prop -> Property
prop_isSat_complete e =
  forAll genSubst $ \s ->
    eval s e == True ==> isSat e /= Nothing
```

Week 2B: Higher-order functions

- note: due to the number of trivial exercises, only complex/useful ones will be included

Deduplication

> The goal of this assignment is to use higher-order functions to implement the function `deduplicate :: (Ord a) => [a] -> [a]` that removes all duplicate elements from a list, leaving only one copy of each element. The order of the elements in the resulting list does not matter. Try to make use of the `Ord` constraint to avoid a quadratic complexity.
>
> **Hint.** Make use of the function `group :: Eq a => [a] -> [[a]]` from the module `Data.List`, which takes a list and returns a list of lists such that the concatenation of the result is equal to the argument. Moreover, each sublist in the result contains only equal elements. For example,

```
>>> group "Mississippi"
["M","i","ss","i","ss","i","pp","i"]
```

Solution:

```
import Data.List

-- This is a quadratic implementation that does not make full use of the Ord constra
deduplicate_slow :: (Ord a) => [a] -> [a]
deduplicate_slow []     = []
deduplicate_slow (x:xs) = x : deduplicate (filter (/= x) xs)

-- Instead, we can make the function faster by first sorting the list.
deduplicate :: (Ord a) => [a] -> [a]
deduplicate = map head . group . sort
```

Spec test:

```
import Data.List

prop_deduplicate_correct :: [Int] -> Property
prop_deduplicate_correct xs = sort (deduplicate xs) === sort (nub xs)
```

Hopscotch

> Implement a function `skips :: [a] -> [[a]]` that outputs a list of lists. The first list in the output should be the input list itself, the second list should consist of every second element of the input list, the third should consist of every third element of the input list, etc. For example:
>
> ```
> > skips [1,2,3,4,5,6]
> [[1,2,3,4,5,6],[2,4,6],[3,6],[4],[5],[6]]
> > skips [True,False]
> skips [[True,False],[False]]
> ```
>
> **Bonus challenge.** Try to find the shortest possible solution by making use of library functions such as `map` and `foldr` .

Solution:

```
-- Here are three possible solutions:

-- Solution 1: using recursion + a helper function
skips' :: [a] -> [[a]]
skips' xs = [skip' i i xs | i <- [0..length xs - 1]]
```

```haskell
  where
    skip' :: Int -> Int -> [a] -> [a]
    skip' n m [] = []
    skip' 0 m (x: xs) = x : skip' m m xs
    skip' n m (x: xs) = skip' (n-1) m xs

-- Solution 2: using a list comprehension + foldr
skips xs = [ foldr (\x f k -> if k==1 then x:f n else f (k-1)) (const []) xs n | n <
```

Spec test:

```haskell
prop_skips_total_length :: [Int] -> Property
prop_skips_total_length xs = length (skips xs) === length xs

prop_skips_individual_lengths :: NonEmptyList Int -> Property
prop_skips_individual_lengths (NonEmpty xs) =
  let n = length xs
  in  forAll (chooseInt (0,n-1)) $ \k ->
      length (skips xs !! k) === n `div` (k+1)

prop_skips_elems :: NonEmptyList Int -> Property
prop_skips_elems (NonEmpty xs) =
  let n = length xs
  in  forAll (chooseInt (0,n-1)) $ \k ->
      let skipsk = skips xs !! k
      in  forAll (chooseInt (0,length skipsk-1)) $ \i ->
          skipsk !! i === xs !! (k+(k+1)*i)
```

Implementing functions

Standard higher order functions

> In this assignment, use each of the following techniques at least once:
>
> - Using a list comprehension
> - Using explicit recursion
> - Using the library function `foldr`
>
> Without looking at the definitions from the standard prelude, define the following higher-order library functions on lists.
>
> - The function `map :: (a -> b) -> [a] -> [b]` applying the given function to each element of the list.
> - The function `filter :: (a -> Bool) -> [a] -> [a]` removing all elements from a list that do not satisfy the given predicate.

> - The function `all :: (a -> Bool) -> [a] -> Bool` deciding if all elements of a list satisfy the given predicate.
> - The function `any :: (a -> Bool) -> [a] -> Bool` deciding if any element of a list satisfies the given predicate.
> - The function `takeWhile :: (a -> Bool) -> [a] -> [a]` selecting all elements from a list until the first element that does not satisfy the given predicate.
> - The function `dropWhile :: (a -> Bool) -> [a] -> [a]` removing all elements from a list until the first element that does not satisfy the given predicate.

Solution:

```
import Prelude hiding (map, filter, all, any, takeWhile, dropWhile)

map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs , p x ]

all :: (a -> Bool) -> [a] -> Bool
all p []     = True
all p (x:xs) = p x && all p xs

any :: (a -> Bool) -> [a] -> Bool
any p = foldr (\x b -> p x || b) False

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p = foldr (\x xs -> if p x then x:xs else []) []

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p []     = []
dropWhile p (x:xs) = if p x then dropWhile p xs else (x:xs)Spec
```

Spec test:

```
import Prelude hiding (map, filter, all, any, takeWhile, dropWhile)

map_type_test :: (a -> b) -> [a] -> [b]
map_type_test = map

prop_map_length :: Fun Int Int -> [Int] -> Property
prop_map_length (Fun _ f) xs = within 1000000 $ length (map f xs) === length xs

prop_map_id :: [Int] -> Property
prop_map_id xs = within 1000000 $ map id xs === xs

prop_map_single :: Fun Int Int -> Int -> Property
```

```haskell
prop_map_single (Fun _ f) x = within 1000000 $ map f [x] === [f x]

filter_type_test :: (a -> Bool) -> [a] -> [a]
filter_type_test = filter

prop_filter_all :: [Int] -> Property
prop_filter_all xs = within 1000000 $ filter (const True) xs === xs

prop_filter_none :: [Int] -> Property
prop_filter_none xs = within 1000000 $ filter (const False) xs === []

prop_filter_empty :: Fun Int Bool -> Property
prop_filter_empty (Fun _ f) = within 1000000 $ filter f [] === []

prop_filter_cons :: Fun Int Bool -> Int -> [Int] -> Property
prop_filter_cons (Fun _ f) x xs = within 1000000 $ filter f (x:xs) === (if f x then

all_type_test :: (a -> Bool) -> [a] -> Bool
all_type_test = all

prop_all_correct :: [Bool] -> Property
prop_all_correct bs = within 1000000 $ all id bs === and bs

any_type_test :: (a -> Bool) -> [a] -> Bool
any_type_test = any

prop_any_correct :: [Bool] -> Property
prop_any_correct bs = within 1000000 $ any id bs === or bs

takeWhile_type_test :: (a -> Bool) -> [a] -> [a]
takeWhile_type_test = takeWhile

prop_takeWhile_ints :: NonNegative Int -> Property
prop_takeWhile_ints (NonNegative n) = within 1000000 $ takeWhile (<= n) [0..] === [0

dropWhile_type_test :: (a -> Bool) -> [a] -> [a]
dropWhile_type_test = dropWhile

prop_dropWhile_ints :: NonNegative Int -> Property
prop_dropWhile_ints (NonNegative n) = within 1000000 $ take 3 (dropWhile (< n+3) [0.
```

Lemon curry

> *Currying* (named after Haskell Curry) is the process of turning a function taking a pair as its argument into a function that takes two separate arguments. Conversely, *uncurrying* is the process of turning a function that takes two separate arguments into a function that takes a pair as its arguments.
>
> For this exercise, reimplement the two standard Haskell functions

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

Solution:

```haskell
import Prelude hiding (curry, uncurry)

curry :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)

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

Spec test:

```haskell
import Prelude hiding (curry, uncurry)
import qualified Test.QuickCheck.Function as Test


test_curry_type :: ((a, b) -> c) -> (a -> b -> c)
test_curry_type = curry

prop_curry_total :: Fun (Int, Int) Int -> Int -> Int -> Property
prop_curry_total (Fun _ f) x y = total $ curry f x y

test_uncurry_type :: (a -> b -> c) -> ((a, b) -> c)
test_uncurry_type = uncurry

prop_uncurry_total :: Fun Int (Fun Int Int) -> (Int, Int) -> Property
prop_uncurry_total (Fun _ f) p = total $ uncurry (\x y -> f x `Test.apply` y) p
```

Folding Expressions

> Consider the type of arithmetic expressions involving `+` and `-`:
>
> ```haskell
> data Expr = Val Int | Add Expr Expr | Subs Expr Expr
> ```
>
> 1. Define a higher-order function `folde :: (Int -> a) -> (a -> a -> a) -> (a -> a -> a) -> Expr -> a` such that `folde f g h` replaces each `Val` constructor in an expression by the function `f`, each `Add` constructor by the function `g`, and each `Subs` constructor with the function `h`.
> 2. Using `folde`, define a function `eval :: Expr -> Int` that evaluates an expression to an integer value.

> 3. Using `folde`, define a function `size :: Expr -> Int` that calculates the number of values in an expression.

Solution:

```
data Expr = Val Int | Add Expr Expr | Subs Expr Expr
  deriving (Show, Eq)

folde :: (Int -> a) -> (a -> a -> a) -> (a -> a -> a) -> Expr -> a
folde f g h (Val x) = f x
folde f g h (Add e1 e2) = g (folde f g h e1) (folde f g h e2)
folde f g h (Subs e1 e2) = h (folde f g h e1) (folde f g h e2)

eval :: Expr -> Int
eval = folde id (+) (-)

size :: Expr -> Int
size = folde (const 1) (+) (+)
```

Spec test:

```
instance Arbitrary Expr where
  arbitrary = sized expr'
    where
      expr' 0       = Val <$> arbitrary
      expr' n | n>0 = oneof [Val <$> arbitrary, Add <$> expr' m <*> expr' m, Subs <$
        where m = n `div` 2

  shrink (Val _)      = []
  shrink (Add e1 e2)  = [e1,e2]
  shrink (Subs e1 e2) = [e1,e2]

folde_type_test :: (Int -> a) -> (a -> a -> a) -> (a -> a -> a) -> Expr -> a
folde_type_test = folde

prop_folde_identity :: Expr -> Property
prop_folde_identity e = within 1000000 $ folde Val Add Subs e === e

prop_eval_val :: Int -> Property
prop_eval_val x = within 1000000 $ eval (Val x) === x

prop_eval_plus :: Int -> Int -> Property
prop_eval_plus x y = within 1000000 $ eval (Add (Val x) (Val y)) === x + y

prop_eval_minus :: Int -> Int -> Property
prop_eval_minus x y = within 1000000 $ eval (Subs (Val x) (Val y)) === x - y

prop_eval_correct :: Expr -> Property
prop_eval_correct e = within 1000000 $ eval e === folde id (+) (-) e
```

```haskell
prop_size_val :: Int -> Property
prop_size_val x = within 1000000 $ size (Val x) === 1

prop_size_plus :: Int -> Int -> Property
prop_size_plus x y = within 1000000 $ size (Add (Val x) (Val y)) === 2

prop_size_minus :: Int -> Int -> Property
prop_size_minus x y = within 1000000 $ size (Subs (Val x) (Val y)) === 2

prop_size_correct :: Expr -> Property
prop_size_correct e = within 1000000 $ size e === folde (const 1) (+) (+) e
```

Unfold

> A higher-order function `unfold` that encapsulates a simple pattern of recursion for producing a list can be defined as follows:
>
> ```haskell
> unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
> unfold p h t x | p x       = []
>                | otherwise = h x : unfold p h t (t x)
> ```
>
> That is, the function `unfold p h t` produces the empty list if the predicate `p` is true of the argument value, and otherwise produces a non-empty list by applying the function `h` to this value to give the head, and the function `t` to generate another argument that is recursively processed in the same way to produce the tail of the list. For example, the function `int2bin` converting an integer to a binary number can be defined using `unfold` as follows:
>
> ```haskell
> int2bin xs = reverse (unfold (== 0) (`mod` 2) (`div` 2) xs)
> ```
>
> Redefine the functions map `:: (a -> b) -> [a] -> [b]` and iterate `:: (a -> a) -> a -> [a]` in terms of `unfold`.

Solution:

```haskell
import Prelude hiding (map, iterate)

map f = unfold null (f . head) tail

iterate f = unfold (const False) id f
```

Spec test:

```haskell
import Prelude hiding (map, iterate)
```

```haskell
import qualified Prelude

map_type_test :: (a -> b) -> [a] -> [b]
map_type_test = map

prop_map_correct :: Fun Int Int -> [Int] -> Property
prop_map_correct (Fun _ f) xs = within 1000000 $ map f xs === Prelude.map f xs

iterate_type_test :: (a -> a) -> a -> [a]
iterate_type_test = iterate

prop_iterate_correct :: Fun Int Int -> Int -> NonNegative Int -> Property
prop_iterate_correct (Fun _ f) a (NonNegative n) = within 1000000 $ take n (iterate
```

Reindexing

> Implement a function `reindex :: (Int -> Int) -> [a] -> [a]` that rearranges the elements of the given list according to the given function: the element of `reindex f xs` at position `f i` should be the same as the element of `xs` at position `i`. In other words, the result should satisfy the equation `reindex f xs !! (f i) == xs !! i`.
>
> For example:
>
> ```
> > reindex id ['h','e','l','l','o']
> ['h','e','l','l','o']
>
> > reindex (\i -> 4-i) [1,2,3,4,5]
> [5,4,3,2,1]
>
> > reindex (\i -> (i+2) `mod` 5) ['a','b','c','d','e']
> ['d','e','a','b','c']
> ```

Solution:

```haskell
reindex :: (Int -> Int) -> [a] -> [a]
reindex f xs = map (\i -> findIndex i ixs) [0..(length xs)-1]
  where
    -- The elements of xs paired with their new indices
    ixs = zip (map f [0..]) xs

    findIndex i ((j,x):jxs)
      | i == j    = x
      | otherwise = findIndex i jxs
```

Spec test:

```haskell
reindex_type_test :: (Int -> Int) -> [a] -> [a]
reindex_type_test = reindex

prop_reindex_id :: [Int] -> Property
prop_reindex_id xs = reindex id xs === xs

prop_reindex_reverse :: [Int] -> Property
prop_reindex_reverse xs = reindex (\i -> length xs - i - 1) xs === reverse xs

prop_reindex_shift :: Property
prop_reindex_shift = forAll (elements [1..100]) $ \n ->
  reindex (\i -> (i - n) `mod` 100) [0..99] === [n..99] ++ [0..(n-1)]
```

Week 3A: Type Classes

> Complete the given instance declarations for the following types:
>
> ```haskell
> data Option a = None | Some a
>
> data List a = Nil | Cons a (List a)
>
> data Tree a = Leaf a | Node (Tree a) a (Tree a)
> ```

Solution:

```haskell
data Option a = None | Some a
  deriving (Show)

data List a = Nil | Cons a (List a)
  deriving (Show)

data Tree a = Leaf a | Node (Tree a) a (Tree a)
  deriving (Show)

instance Eq a => Eq (Option a) where
  None   == None   = True
  Some x == Some y = x == y
  _      == _      = False

instance Eq a => Eq (List a) where
  Nil         == Nil         = True
  (Cons x xs) == (Cons y ys) = x == y && xs == ys
  _           == _           = False

instance Eq a => Eq (Tree a) where
  (Leaf x)     == (Leaf y)       = x == y
```

```
  (Node l x r) == (Node l' x' r') = l == l' && x == x' && r == r'
  _            == _               = False
```

Spec test:

```
instance Arbitrary a => Arbitrary (Option a) where
  arbitrary = oneof [pure None, Some <$> arbitrary]

  shrink _ = []

instance Arbitrary a => Arbitrary (List a) where
  arbitrary = sized list'
    where
      list' 0       = pure Nil
      list' n | n>0 = oneof [pure Nil, Cons <$> arbitrary <*> list' m]
        where m = n `div` 2

  shrink (Nil)       = []
  shrink (Cons x xs) = [xs] ++ [Cons x xs' | xs' <- shrink xs]

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized tree'
    where
      tree' 0       = oneof [Leaf <$> arbitrary]
      tree' n | n>0 = oneof [Leaf <$> arbitrary, Node <$> tree' m <*> arbitrary <*>
        where m = n `div` 2

  shrink (Leaf x)     = [Leaf x' | x' <- shrink x]
  shrink (Node l x r) = [Leaf x,l,r] ++ [Node l' x' r' | (l',x',r') <- shrink (l,x,r

prop_eq_option_refl :: Option Int -> Property
prop_eq_option_refl x = within 1000000 $ x == x

prop_eq_option_some_none :: Int -> Property
prop_eq_option_some_none x = within 1000000 $ not $ Some x == None

prop_eq_option_none_some :: Int -> Property
prop_eq_option_none_some x = within 1000000 $ not $ None == Some x

prop_eq_list_refl :: List Int -> Property
prop_eq_list_refl x = within 1000000 $ x == x

prop_eq_list_shrink :: Int -> List Int -> Property
prop_eq_list_shrink x xs = forAll (elements (shrink (Cons x xs))) $ \xs' -> not $ Co

prop_eq_tree_refl :: Tree Int -> Property
prop_eq_tree_refl x = within 1000000 $ x == x

prop_eq_tree_shrink :: Tree Int -> Int -> Tree Int -> Property
prop_eq_tree_shrink l x r =
```

```
let t = Node l x r in
forAll (elements (shrink (Node l x r))) $ \t' ->
counterexample (show t ++ "\n should not be equal to \n" ++ show t' ++ "!") $
not (Node l x r == t') .&. not (t' == Node l x r)
```

Unary natural numbers

> The recursive type of (unary) natural numbers is defined as follows:
>
> ```
> data Nat = Zero | Suc Nat
> ```
>
> **Assignment 1.** Define a recursive function `natToInteger :: Nat -> Integer` that converts a unary natural number to a Haskell `Integer` .
>
> **Assignment 2.** Define the recursive functions `add :: Nat -> Nat -> Nat` , `mult :: Nat -> Nat -> Nat` , and `pow :: Nat -> Nat -> Nat` .
>
> Hint: make use of the functions you already defined.
>
> **Assignment 3.** Use a `deriving` clause to automatically define instances of the `Show` and `Eq` typeclasses for the `Nat` type.
>
> **Assignment 4.** Define an instance of the `Ord` typeclass for the `Nat` type. The instance declaration should look as follows:
>
> ```
> instance Ord Nat where
>   -- (<=) :: Nat -> Nat -> Bool
>   x <= y = ...
> ```
>
> **Assignment 5.** Define an instance of the `Num` typeclass for the `Nat` type. The instance declaration should look as follows:
>
> ```
> instance Num Nat where
>   -- (+) :: Nat -> Nat -> Nat
>   x + y = ...
>
>   -- (*) :: Nat -> Nat -> Nat
>   x * y = ...
>
>   -- fromInteger :: Integer -> Nat
>   fromInteger x = ...
> ```
>
> You do not need to give definitions for the functions `abs` , `signum` , and `negate` . **Hint.** Make use of the `add` and `mult` functions you defined before.

Solution:

```
data Nat = Zero | Suc Nat
  deriving (Show, Eq)

natToInteger :: Nat -> Integer
natToInteger Zero    = 0
natToInteger (Suc n) = 1 + natToInteger n

add :: Nat -> Nat -> Nat
add Zero n = n
add (Suc m) n = Suc (add m n)

mult :: Nat -> Nat -> Nat
mult Zero n = Zero
mult (Suc m) n = add n (mult m n)

pow :: Nat -> Nat -> Nat
pow m Zero = Suc Zero
pow m (Suc n) = mult m (pow m n)

instance Ord Nat where
  -- (<=) :: Nat -> Nat -> Bool
  Zero <= y          = True
  (Suc x) <= Zero    = False
  (Suc x) <= (Suc y) = x <= y

instance Num Nat where
  -- (+) :: Nat -> Nat -> Nat
  x + y = add x y

  -- (*) :: Nat -> Nat -> Nat
  x * y = mult x y

  --- fromInteger :: Integer -> Nat
  fromInteger x
    | x == 0    = Zero
    | x > 0     = Suc (fromInteger (x-1))
    | otherwise = undefined
```

Spec test:

```
natToInteger_spec :: Nat -> Integer
natToInteger_spec Zero = 0
natToInteger_spec (Suc n) = 1 + natToInteger n

instance Arbitrary Nat where
  arbitrary = sized nat'
    where
```

```
      nat' 0       = pure Zero
      nat' n | n>0 = oneof [pure Zero, Suc <$> (nat' (n `div` 2))]

  shrink Zero = []
  shrink (Suc n) = [n]

prop_natToInteger_correct :: Nat -> Property
prop_natToInteger_correct n = natToInteger n === natToInteger_spec n

prop_add_correct :: Nat -> Nat -> Property
prop_add_correct m n = natToInteger_spec (add m n) === natToInteger_spec m + natToIr

prop_mult_correct :: Nat -> Nat -> Property
prop_mult_correct m n = natToInteger_spec (mult m n) === natToInteger_spec m * natTc

prop_pow_correct :: Nat -> Nat -> Property
prop_pow_correct m n = natToInteger_spec (pow m n) === natToInteger_spec m ^ natToIr

prop_leq_correct :: Nat -> Nat -> Property
prop_leq_correct m n = (m <= n) === (natToInteger_spec m <= natToInteger_spec n)

plus_type_test :: Nat -> Nat -> Nat
plus_type_test = (+)

mult_type_test :: Nat -> Nat -> Nat
mult_type_test = (*)

prop_fromInteger_correct :: Nat -> Property
prop_fromInteger_correct n = fromInteger (natToInteger_spec n) === n
```

ABBA

> The Haskell type class `Semigroup` is defined as follows:
>
> ```
> class Semigroup a where
>   (<>) :: a -> a -> a
> ```
>
> It has instances for many types, including lists the `Sum` type:
>
> ```
> instance Semigroup [a] where
>   (<>) = (++)
>
> newtype Sum a = Sum a
>
> instance Num a => Semigroup (Sum a) where
>     Sum x <> Sum y = Sum (x + y)
> ```

> Define a function `abba` that takes two argument of some type `a` which implements the `Semigroup` class. The result should be the two items appended in an “ABBA” pattern:
>
> ```
> abba [1,2] [3] = [1,2,3,3,1,2]
> abba (Sum 2) (Sum 3) = Sum (2+3+3+2) = Sum 10
> ```

Solution:

```
import Data.Monoid

abba :: Semigroup a => a -> a -> a
abba a b = a <> b <> b <> a
```

Spec test:

```
import Data.Monoid

prop_abba_list xs ys = abba (xs :: [Int]) ys === xs ++ ys ++ ys ++ xs

prop_abba_maybe_list xs ys = abba (xs :: Maybe [Int]) ys === xs <> ys <> ys <> xs

prop_abba_sum x y = abba (x :: Sum Int) y === x <> y <> y <> x

prop_abba_product x y = abba (x :: Product Int) y === x <> y <> y <> x
```

Shapes

> Define a new typeclass `Shape a` with the following functions:
>
> - `corners :: a -> Int`
> - `circumference :: a -> Double`
> - `surface :: a -> Double`
> - `rescale :: Double -> a -> a`
>
> Then, define instances of this typeclass for the following types:
>
> ```
> data Square = Square { squareSide :: Double }
>
> data Rectangle = Rect { rectWidth :: Double , rectHeight :: Double }
>
> data Circle = Circle { circleRadius :: Double }
> ```
>
> *Bonus:* also implement instances for the following types:

> ```
> data Triangle = Triangle { triangleSide1 :: Double, triangleSide2 :: Double, tr
>
> data RegularPolygon = Poly { polySides :: Int , polySideLength :: Double }
> ```

Solution:

```
data Square = Square { squareSide :: Double }
  deriving (Show, Eq)
data Rectangle = Rect { rectWidth :: Double , rectHeight :: Double }
  deriving (Show, Eq)
data Circle = Circle { circleRadius :: Double }
  deriving (Show, Eq)
data Triangle = Triangle { triangleSide1 :: Double, triangleSide2 :: Double, triangl
  deriving (Show, Eq)
data RegularPolygon = Poly { polySides :: Int , polySideLength :: Double }
  deriving (Show, Eq)

class Shape a where
  corners       :: a -> Int
  circumference :: a -> Double
  surface       :: a -> Double
  rescale       :: Double -> a -> a

instance Shape Square where
  corners       _          = 4
  circumference (Square x) = 4*x
  surface       (Square x) = x*x
  rescale s     (Square x) = Square (s*x)

instance Shape Rectangle where
  corners       _          = 4
  circumference (Rect x y) = 2*x + 2*y
  surface       (Rect x y) = x*y
  rescale s     (Rect x y) = Rect (s*x) (s*y)

instance Shape Circle where
  corners       _          = 0
  circumference (Circle r) = 2*pi*r
  surface       (Circle r) = pi*r*r
  rescale s     (Circle r) = Circle (s*r)

instance Shape Triangle where
  corners       _                = 3
  circumference (Triangle x y z) = x+y+z
  surface       (Triangle x y z) = sqrt (s*(s-x)*(s-y)*(s-z))
    where s = (x+y+z)/2
  rescale s     (Triangle x y z) = Triangle (s*x) (s*y) (s*z)

instance Shape RegularPolygon where
```

```haskell
  corners       (Poly n x) = n
  circumference (Poly n x) = (fromIntegral n)*x
  surface       (Poly n x) = (fromIntegral n)*x*x/(4*tan(pi/(fromIntegral n)))
  rescale s     (Poly n x) = Poly n (s*x)
```

Spec test:

```haskell
arbitrarySide = (2+) . getPositive <$> arbitrary

instance Arbitrary Square where
  arbitrary = Square <$> arbitrarySide

instance Arbitrary Rectangle where
  arbitrary = Rect <$> arbitrarySide <*> arbitrarySide

instance Arbitrary Circle where
  arbitrary = Circle <$> arbitrarySide

instance Arbitrary Triangle where
  arbitrary = do
    (x,y,z) <- (,,) <$> arbitrarySide <*> arbitrarySide <*> arbitrarySide
    if x >= y+z || y >= x+z || z >= x+y then
      return discard
    else
      return $ Triangle x y z

instance Arbitrary RegularPolygon where
  arbitrary = Poly <$> (getPositive <$> arbitrary) <*> arbitrarySide

epsilon = 0.01

class Eq a => Approx a where
  approx :: a -> a -> Property

instance Approx Double where
  x `approx` y = x === y .||. abs (x - y) / max (abs x) (abs y) < epsilon

instance Approx Square where
  (Square x) `approx` (Square y) = x `approx` y

instance Approx Rectangle where
  (Rect x y) `approx` (Rect z w) = (x `approx` z) .&&. (y `approx` w)

instance Approx Circle where
  (Circle x) `approx` (Circle y) = x `approx` y

instance Approx Triangle where
  (Triangle x y z) `approx` (Triangle u v w) = (x `approx` u) .&&. (y `approx` v) .&
```

```haskell
instance Approx RegularPolygon where
  (Poly m x) `approx` (Poly n y) = m === n .&&. x `approx` y

prop_square_corners :: Square -> Property
prop_square_corners x = corners x === 4

prop_square_circumference :: Square -> Property
prop_square_circumference (Square x) = (circumference (Square x)) `approx` (4*x)

prop_square_surface :: Square -> Property
prop_square_surface (Square x) = (surface (Square x)) `approx` (x*x)

prop_square_rescale :: Positive Double -> Square -> Property
prop_square_rescale (Positive s) (Square x) = (rescale s (Square x)) `approx` (Squar

prop_rect_corners :: Rectangle -> Property
prop_rect_corners x = corners x === 4

prop_rect_circumference :: Rectangle -> Property
prop_rect_circumference (Rect x y) = (circumference (Rect x y)) `approx` (2*x + 2*y)

prop_rect_surface :: Rectangle -> Property
prop_rect_surface (Rect x y) = (surface (Rect x y)) `approx` (x*y)

prop_rect_rescale :: Positive Double -> Rectangle -> Property
prop_rect_rescale (Positive s) (Rect x y) = (rescale s (Rect x y)) `approx` (Rect (s

prop_circle_corners :: Circle -> Property
prop_circle_corners x = corners x === 0

prop_circle_circumference :: Circle -> Property
prop_circle_circumference (Circle r) = (circumference (Circle r)) `approx` (2*pi*r)

prop_circle_surface :: Circle -> Property
prop_circle_surface (Circle r) = (surface (Circle r)) `approx` (pi*r*r)

prop_circle_rescale :: Positive Double -> Circle -> Property
prop_circle_rescale (Positive s) (Circle r) = (rescale s (Circle r)) `approx` (Circl

prop_triangle_corners :: Triangle -> Property
prop_triangle_corners x = corners x === 3

prop_triangle_circumference :: Triangle -> Property
prop_triangle_circumference (Triangle x y z) = (circumference (Triangle x y z)) `app

prop_triangle_surface :: Triangle -> Property
prop_triangle_surface (Triangle x y z) = (surface (Triangle x y z)) `approx` (sqrt (
  where s = (x+y+z)/2

prop_triangle_rescale :: Positive Double -> Triangle -> Property
prop_triangle_rescale (Positive s) (Triangle x y z) = (rescale s (Triangle x y z)) `
```

```
prop_poly_corners :: RegularPolygon -> Property
prop_poly_corners (Poly n x) = corners (Poly n x) === n

prop_poly_circumference :: RegularPolygon -> Property
prop_poly_circumference (Poly n x) = (circumference (Poly n x)) `approx` ((fromInteg

prop_poly_surface :: RegularPolygon -> Property
prop_poly_surface (Poly n x) = n>=3 ==> (surface (Poly n x)) `approx` ((fromIntegral

prop_poly_rescale :: Positive Double -> RegularPolygon -> Property
prop_poly_rescale (Positive s) (Poly n x) = (rescale s (Poly n x)) `approx` (Poly n
```

Quaternions

> Quaternions are a generalization of complex numbers developed initially by the Irish mathematician Hamilton to solve dynamics problems in physics. More recently, they have been used in computer graphics to efficiently compute transformations in 3D space. Where complex numbers have two components (a real and an imaginary part), quaternions have four. An arbitrary quaternion can be written as `a + b*i + c*j + d*k` where `a,b,c,d` are real numbers and `i,j,k` are constants satisfying the following laws:
>
> - `i*i = -1`
> - `j*j = -1`
> - `k*k = -1`
> - `i*j = k`
> - `j*i = -k`
> - `j*k = i`
> - `k*j = -i`
> - `k*i = j`
> - `i*k = -j`
>   Note that multiplication on quaternions is not commutative: `i*j` is not equal to `j*i` !
>
> Your task is to implement a Haskell type `Quaternion` and define the constants `i,j,k :: Quaternion`, a function `fromDouble :: Double -> Quaternion`, and give instances for the `Eq`, `Show`, and `Num` classes. Some further details:
>
> - Quaternions should be pretty-printed in the format `1.2 + 3.4i + 5.6j + 7.8k`
> - The absolute value of a quaternion equals the square root of the sum of the squares of all its components, i.e. *abs*(*a*+*bi*+*cj*+*dk*)=sqrt(*a*^2+*b*^2+*c*^2+*d*^2)
>
> - The `abs` and `signum` functions should satisfy the equation `x = abs x * signum x` for any quaternion `x`.

Solution:

```haskell
data Quaternion = Q Double Double Double Double
  deriving Eq

-- Take the real part of a quaternion (used for testing)
-- realPart (a + b*i + c*j + d*k) == a
realPart :: Quaternion -> Double
realPart (Q a _ _ _) = a


i, j, k :: Quaternion
i = Q 0 1 0 0
j = Q 0 0 1 0
k = Q 0 0 0 1

fromDouble :: Double -> Quaternion
fromDouble x = Q x 0 0 0

instance Show Quaternion where
  show (Q a b c d) = show a ++ " + " ++ show b ++ "i + " ++ show c ++ "j + " ++ show

instance Num Quaternion where
  (Q a b c d) + (Q e f g h) = Q (a+e) (b+f) (c+g) (d+h)
  (Q a b c d) * (Q e f g h) = Q (a*e-b*f-c*g-d*h) (a*f+b*e+c*h-d*g) (a*g-b*h+c*e+d*f
  abs (Q a b c d) = Q (sqrt (a^2+b^2+c^2+d^2)) 0 0 0
  signum (Q a b c d) = Q (a/e) (b/e) (c/e) (d/e)
    where e = sqrt (a^2+b^2+c^2+d^2)
  negate (Q a b c d) = Q (negate a) (negate b) (negate c) (negate d)
  fromInteger n = Q (fromInteger n) 0 0 0
```

Spec test:

```haskell
instance Arbitrary Quaternion where
  arbitrary = (\a b c d -> fromDouble a + fromDouble b * i + fromDouble c * j + from

prop_show_quaternion :: Double -> Double -> Double -> Double -> Property
prop_show_quaternion a b c d =
  show (fromDouble a + fromDouble b * i + fromDouble c * j + fromDouble d * k)
  === show a ++ " + " ++ show b ++ "i + " ++ show c ++ "j + " ++ show d ++ "k"

prop_real_nonzero :: NonZero Double -> Property
prop_real_nonzero (NonZero x) = fromDouble x =/= fromDouble 0

prop_i_nonzero :: NonZero Double -> Property
prop_i_nonzero (NonZero x) = fromDouble x * i =/= fromDouble 0

prop_j_nonzero :: NonZero Double -> Property
prop_j_nonzero (NonZero x) = fromDouble x * j =/= fromDouble 0
```

```haskell
prop_k_nonzero :: NonZero Double -> Property
prop_k_nonzero (NonZero x) = fromDouble x * k =/= fromDouble 0

prop_add_quaternion_zero :: Quaternion -> Property
prop_add_quaternion_zero x = 0 + x === x

prop_add_quaternion_neg :: Quaternion -> Property
prop_add_quaternion_neg x = x + negate x === fromInteger 0

prop_add_quaternion_comm :: Quaternion -> Quaternion -> Property
prop_add_quaternion_comm x y = x + y === y + x

prop_mult_i_i :: Property
prop_mult_i_i = i*i === fromInteger (-1)

prop_mult_j_j :: Property
prop_mult_j_j = j*j === fromInteger (-1)

prop_mult_k_k :: Property
prop_mult_k_k = k*k === fromInteger (-1)

prop_mult_i_j :: Property
prop_mult_i_j = i*j === k

prop_mult_i_k :: Property
prop_mult_i_k = i*k === -j

prop_mult_j_i :: Property
prop_mult_j_i = j*i === -k

prop_mult_j_k :: Property
prop_mult_j_k = j*k === i

prop_mult_k_i :: Property
prop_mult_k_i = k*i === j

prop_mult_k_j :: Property
prop_mult_k_j = k*j === -i

epsilon :: Double
epsilon = 0.01

diff :: Quaternion -> Quaternion -> Double
diff x y = realPart (x - y)

prop_abs :: Double -> Double -> Double -> Double -> Bool
prop_abs a b c d =
  diff (abs (fromDouble a + (fromDouble b)*i + (fromDouble c)*j + (fromDouble d)*k))
       (fromDouble (sqrt (a*a + b*b + c*c + d*d)))
       < epsilon
```

```
check_diff :: Quaternion -> Quaternion -> Double
check_diff x y = realPart (abs (x - y))

prop_abs_signum :: Quaternion -> Property
prop_abs_signum x = x /= fromDouble 0 ==> check_diff x (abs x * signum x) < epsilon
```

Pretty printing JSON data

The JSON (JavaScript Object Notation) language is a small, simple representation for storing and transmitting structured data, for example over a network connection. It is most commonly used to transfer data from a web service to a browser-based JavaScript application. The JSON format is described at www.json.org, and in greater detail by RFC 4627.

JSON supports four basic types of value: strings, numbers, booleans, and a special value named null. The language provides two compound types: an array is an ordered sequence of values, and an object is an unordered collection of name/value pairs. The names in an object are always strings; the values in an object or array can be of any type.

To work with JSON data in Haskell, we use an algebraic data type to represent the range of possible JSON types.

Exercise 1. Define a datatype `JValue` with constructors `JString` (storing a `String`), `JNumber` (storing a `Double`), `JBool` (storing a `Bool`), `JNull` , `JObject` (storing a list of key-value pairs), and `JArray` (storing a list of values). Add `deriving Show` to the end of your definition to derive a `Show` instance for your type.

Exercise 2. Implement an instance of the `Eq` class for `JValue` .

We can see how to use a constructor to take a normal Haskell value and turn it into a JValue. To do the reverse, we use pattern matching.

Exercise 3. Implement the following functions for converting JSON values to Haskell values:

- `getString :: JValue -> Maybe String`
- `getInt :: JValue -> Maybe Int`
- `getDouble :: JValue -> Maybe Double`
- `getBool :: JValue -> Maybe Bool`
- `getObject :: JValue -> Maybe [(String, JValue)]`
- `getArray :: JValue -> Maybe [JValue]`
- `isNull :: JValue -> Bool`

Hint. The function `getInt` should round the given number down to the nearest integer. For this, you can use the function `truncate` .

> Now that we have a Haskell representation for JSON's types, we'd like to be able to take Haskell values and render them as JSON data.
>
> **Exercise 4.** Implement a function `renderJValue :: JValue -> String` that prints a value in JSON form (see the "Test" tab for some examples).
>
> Note that when pretty printing a string value, JSON has moderately involved escaping rules that we must follow. For this exercise, you can approximate the escaping rules by using `show` on the string. This will use the Haskell escaping rules rather than the JSON escaping rules, which is good enough for the tests of this exercise. For the full project you will need to implement the proper JSON escaping rules, however.
>
> (This assignment is based on the material from [Chapter 5 of Real World Haskell](), which is licensed under a Attribution-NonCommercial 3.0 Unported Creative Commons license.)

Solution:

```haskell
import Data.List (intercalate)

data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject [(String, JValue)]
            | JArray [JValue]
              deriving (Eq, Show)

getInt (JNumber n) = Just (truncate n)
getInt _           = Nothing

getDouble (JNumber n) = Just n
getDouble _           = Nothing

getBool (JBool b) = Just b
getBool _         = Nothing

getObject (JObject o) = Just o
getObject _           = Nothing

getArray (JArray a) = Just a
getArray _          = Nothing

isNull v            = v == JNull



renderJValue :: JValue -> String

renderJValue (JString s)   = show s
```

```haskell
renderJValue (JNumber n)    = show n
renderJValue (JBool True)  = "true"
renderJValue (JBool False) = "false"
renderJValue JNull         = "null"

renderJValue (JObject o) = "{" ++ pairs o ++ "}"
  where pairs [] = ""
        pairs ps = intercalate ", " (map renderPair ps)
        renderPair (k,v)   = show k ++ ": " ++ renderJValue v

renderJValue (JArray a) = "[" ++ values a ++ "]"
  where values [] = ""
        values vs = intercalate ", " (map renderJValue vs)
```

Spec test:

```haskell
import Data.List (intercalate)

{-
data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject [(String, JValue)]
            | JArray [JValue]
-}

instance Arbitrary JValue where
  arbitrary = sized val'
    where
      val' 0       = oneof baseCases
      val' n | n>0 = oneof (baseCases ++ recCases (n `div` 2))

      baseCases = [ JString <$> arbitrary
                  , JNumber <$> arbitrary
                  , JBool <$> arbitrary
                  , pure JNull
                  ]
      recCases m = [ JObject <$> resize m (listOf ((,) <$> arbitrary <*> val' m))
                   , JArray <$> resize m (listOf (val' m))
                   ]

  shrink (JString _) = []
  shrink (JNumber _) = []
  shrink (JBool   _) = []
  shrink JNull       = []
  shrink (JObject xs) = map JObject (shrink xs) ++ map snd xs
  shrink (JArray xs)  = map JArray (shrink xs) ++ xs

prop_getInt_number n = within 1000 $ getInt (JNumber n) === Just (truncate n)
```

```haskell
prop_getDouble_number n = within 1000 $ getDouble (JNumber n) === Just n

prop_getBool_bool b = within 1000 $ getBool (JBool b) === Just b

prop_getObject_obj o = within 1000000 $ getObject (JObject o) === Just o

prop_getArray_arr a = within 1000000 $ getArray (JArray a) === Just a

prop_isNull_null = within 1000 $ isNull JNull

prop_render_string s = renderJValue (JString s) === show s
prop_render_number n = renderJValue (JNumber n) === show n
prop_render_true = renderJValue (JBool True)  === "true"
prop_render_false = renderJValue (JBool False) === "false"
prop_render_null = renderJValue JNull === "null"
prop_render_object o = renderJValue (JObject o) === "{" ++ pairs o ++ "}"
  where pairs [] = ""
        pairs ps = intercalate ", " (map renderPair ps)
        renderPair (k,v)   = show k ++ ": " ++ renderJValue v
prop_render_array a = renderJValue (JArray a) === "[" ++ values a ++ "]"
  where values [] = ""
        values vs = intercalate ", " (map renderJValue vs)
```

Week 3B: Functors

Using Functors

> A functor is a type constructor that has an operation `fmap` with the following signature:
>
> ```haskell
> class Functor f where
>   fmap :: (a -> b) -> f a -> f b
> ```
>
> Applying `fmap g` to a value `x :: f a` applies the function `g` to all values of type `a` that are stored inside `x`.
>
> You are given the definition of a function `multiplySqrtDouble`. Rewrite this function so that it uses `fmap` instead of a case statement. Try to get the definition on one, short line!

Solution:

```haskell
safeSquareRoot :: Double -> Maybe Double
safeSquareRoot x = if x < 0 then Nothing else Just (sqrt x)
```

```haskell
multiplySqrtDouble :: Double -> Double -> Maybe Double
multiplySqrtDouble x y = fmap (*2) (safeSquareRoot (x * y))
```

Spec test:

```haskell
prop_safeSquareRoot_nothing1 (Negative x) (Positive y) = multiplySqrtDouble (x :: Do

prop_safeSquareRoot_nothing2 (Positive x) (Negative y) = multiplySqrtDouble (x :: Do

prop_safeSquareRoot_just1 (Positive x) (Positive y) =
  case multiplySqrtDouble (x :: Double) y of
    Nothing -> False
    Just z  -> abs (z - (sqrt (x * y) * 2)) < 0.001

prop_safeSquareRoot_just2 (Negative x) (Negative y) =
  case multiplySqrtDouble (x :: Double) y of
    Nothing -> False
    Just z  -> abs (z - (sqrt (x * y) * 2)) < 0.001
```

Double all the metrics

> You are given the following datatype to collect a set metrics:
>
> ```haskell
> data Metrics m = Metrics
>   { latestMeasurements :: [m]
>   , average :: m
>   , max :: m
>   , min :: m
>   , mode :: Maybe m
>   } deriving (Show, Eq)
> ```
>
> First, write a `Functor` instance for this datatype. After that, implement a simple function `doubleMetrics` that doubles the values of all the metrics, by using `fmap`.
>
> > Note: if you want to use the `min` and `max` function from the `Metrics` data type, you can add `import Prelude hiding (min, max)`.

Solution:

```haskell
data Metrics m = Metrics
  { latestMeasurements :: [m]
  , average :: m
  , max :: m
  , min :: m
  , mode :: Maybe m
  } deriving (Show, Eq)
```

```haskell
instance Functor Metrics where
  fmap f (Metrics xs a b c d) = Metrics (fmap f xs) (f a) (f b) (f c) (fmap f d)

doubleMetrics :: Metrics Double -> Metrics Double
doubleMetrics = fmap (*2)
```

Spec test:

```haskell
prop_functor_id xs a b c d = fmap id (Metrics xs a b c d :: Metrics Int) === Metrics

prop_doubleMetrics_correct xs a b c d =
  doubleMetrics (Metrics xs a b c d) === Metrics (map (*2) xs) (a*2) (b*2) (c*2) (fm
```

Functor Tree

> Define an `instance` of the `Functor` class for the following type of binary trees that have data in their nodes:
>
> ```haskell
> data Tree a = Leaf | Node (Tree a) a (Tree a)
> ```
>
> For example, `fmap (*2) (Node Leaf 1 Leaf)` should return `Node Leaf 2 Leaf` .

Solution:

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a)
  deriving (Show, Eq)

instance Functor Tree where
  fmap f Leaf = Leaf
  fmap f (Node l x r) = Node (fmap f l) (f x) (fmap f r)
```

Spec test:

```haskell
fmap_type_test :: (a -> b) -> Tree a -> Tree b
fmap_type_test = fmap

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized tree'
    where
      tree' 0       = pure Leaf
      tree' n | n>0 = oneof [pure Leaf, Node <$> tree' m <*> arbitrary <*> tree' m]
        where m = n `div` 2

  shrink Leaf     = []
```

```haskell
  shrink (Node l x r) = [Leaf,l,r] ++ [Node l' x' r' | (l',x',r') <- shrink (l,x,r)]

prop_fmap_single :: Fun Int Int -> Int -> Property
prop_fmap_single (Fun _ f) x = fmap f (Node Leaf x Leaf) === Node Leaf (f x) Leaf

prop_fmap_node :: Tree Int -> Int -> Tree Int -> Bool
prop_fmap_node l x r = isNode (fmap id (Node l x r))
  where
    isNode Leaf{} = False
    isNode Node{} = True

prop_fmap_id :: Tree Int -> Property
prop_fmap_id t = fmap id t === t
```

A Functor of Expressions

> Consider the following type `Expr a` of arithmetic expressions that contain variables of some type `a`:
>
> ```haskell
> data Expr a = Var a | Val Int | Add (Expr a) (Expr a)
>   deriving (Show, Eq)
> ```
>
> For example, if we want to represent variables as string we can use the type `Expr String`. Show how to make this type into an instance of the `Functor` class.

Solution:

```haskell
data Expr a = Var a | Val Int | Add (Expr a) (Expr a)
  deriving (Show, Eq)

instance Functor Expr where
  -- fmap :: (a -> b) -> Expr a -> Expr b
  fmap f (Var x) = Var (f x)
  fmap f (Val i) = Val i
  fmap f (Add p q) = Add (fmap f p) (fmap f q)
```

Spec test:

```haskell
instance Arbitrary a => Arbitrary (Expr a) where
  arbitrary = sized expr'
    where
      expr' 0       = oneof [ Var <$> arbitrary, Val <$> arbitrary ]
      expr' n | n>0 = oneof [ Var <$> arbitrary
                            , Val <$> arbitrary
                            , Add <$> expr' m <*> expr' m
                            ]
```

```haskell
        where m = n `div` 2

  shrink (Var x) = map Var $ shrink x
  shrink (Val x) = map Val $ shrink x
  shrink (Add x y) = [x,y] ++ [Add x y' | y' <- shrink y ] ++ [ Add x' y | x' <- shr

fmap_type_test :: (a -> b) -> Expr a -> Expr b
fmap_type_test = fmap

prop_fmap_var :: Fun Int Int -> Int -> Property
prop_fmap_var (Fun _ f) x = fmap f (Var x) === Var (f x)

prop_fmap_val :: Fun Int Int -> Int -> Property
prop_fmap_val (Fun _ f) x = fmap f (Val x) === Val x

prop_fmap_same_constructor :: Fun String String -> Expr String -> Bool
prop_fmap_same_constructor (Fun _ f) e = sameCon (fmap f e) e
  where
    sameCon Var{} Var{} = True
    sameCon Val{} Val{} = True
    sameCon Add{} Add{} = True
    sameCon _     _     = False

prop_fmap_id :: Expr Int -> Property
prop_fmap_id e = fmap id e === e
```

Using applicatives (1)

> An applicative functor is a functor that has two additional operations `pure` and `(<*>)` with the following signatures:
>
> ```haskell
> class Functor f => Applicative f where
>   pure :: a -> f a
>   (<*>) :: f (a -> b) -> f a -> f b
> ```
>
> - The `pure` function tells us how to wrap an element in the structure in the most basic way.
> - The `<*>` function is the *apply operator* and takes a transformation within the structure, the structure containing the first type, and performs the transformation over the whole structure.
>
> For example, `pure 1` returns `Just 1`, `pure (*2) <*> Just 1` returns `Just 2` and `pure (*2) <*> Nothing` returns `Nothing`.
>
> Using these two operations and the given function `safeSquareRoot`, implement the function `sumOfSquareRoots` that returns the sum of the square roots of the two inputs.

> **Note.** Your solution should *not* make explicit use of the `Nothing` and `Just` constructors of the `Maybe` type, only the `safeSquareRoot` function and the operators of the `Applicative` class.

Solution:

```
sumOfSquareRoots :: Double -> Double -> Maybe Double
sumOfSquareRoots x y = pure (+) <*> safeSquareRoot x <*> safeSquareRoot y
```

Spec test:

```
prop_sumOfSquareRoots_neg1 :: Negative Double -> Double -> Property
prop_sumOfSquareRoots_neg1 (Negative x) y = sumOfSquareRoots x y === Nothing

prop_sumOfSquareRoots_neg2 :: Double -> Negative Double -> Property
prop_sumOfSquareRoots_neg2 x (Negative y) = sumOfSquareRoots x y === Nothing

prop_sumOfSquareRoots_pospos :: Positive Double -> Positive Double -> Property
prop_sumOfSquareRoots_pospos (Positive x) (Positive y) = sumOfSquareRoots x y === Ju
```

Using Applicatives (2)

> Using the two operations `pure` and `(<*>)` of the `Applicative` type class, implement the function `generateAllResults` that takes a list of operations and two lists of numbers and returns a list of all combinations of these operations applied to one element of the first list and one element of the second list. For example:
>
> ```
> generateAllResults [(+)] [1,2] [10,20] = [11,21,12,22]
> generateAllResults [(+),(*)] [1,2] [3,4] = [4, 5, 5, 6, 3, 4, 6, 8]
> generateAllResults [(+), (*), (-)] [10] [3, 4, 5] = [13, 14, 15, 30, 40, 50, 7,
> ```
>
> **Note.** Your solution should *not* use a list comprehension, only the operators of the `Applicative` class. Try to make your solution fit on a single short line!

Solution:

```
prop_example1 = generateAllResults [(+)] [1,2] [10,20] === [11,21,12,22]
prop_example2 = generateAllResults [(+),(*)] [1,2] [3,4] === [4, 5, 5, 6, 3, 4, 6, 8
prop_example3 = generateAllResults [(+), (*), (-)] [10] [3, 4, 5] === [13, 14, 15, 3

-- Testing with a single operation +
prop_generateSums xs ys = generateAllResults [(+)] xs ys === [ x+y | x <- xs, y <- y

-- Testing with an arbitrary subset of {+,-,*}
```

```
prop_generateSumsAndProducts xs ys =
  forAllShow (sublistOf [("+",(+)),("-",(-)),("*",(*))]) (show . map fst) $ \fs ->
  let fs' = map snd fs in
  generateAllResults fs' xs ys === [ f x y | f <- fs', x <- xs, y <- ys ]

-- Testing with a list of arbitrary functions generated by QuickCheck
-- Only look at the first 10 elements to avoid tests timeout
prop_generateAllResults fs xs ys =
  let fs' = map applyFun2 fs in
  take 10 (generateAllResults fs' xs ys) === take 10 [ f x y | f <- fs', x <- xs, y
```

Spec tests:

```
prop_example1 = generateAllResults [(+)] [1,2] [10,20] === [11,21,12,22]
prop_example2 = generateAllResults [(+),(*)] [1,2] [3,4] === [4, 5, 5, 6, 3, 4, 6, 8
prop_example3 = generateAllResults [(+), (*), (-)] [10] [3, 4, 5] === [13, 14, 15, 3

-- Testing with a single operation +
prop_generateSums xs ys = generateAllResults [(+)] xs ys === [ x+y | x <- xs, y <- y

-- Testing with an arbitrary subset of {+,-,*}
prop_generateSumsAndProducts xs ys =
  forAllShow (sublistOf [("+",(+)),("-",(-)),("*",(*))]) (show . map fst) $ \fs ->
  let fs' = map snd fs in
  generateAllResults fs' xs ys === [ f x y | f <- fs', x <- xs, y <- ys ]

-- Testing with a list of arbitrary functions generated by QuickCheck
-- Only look at the first 10 elements to avoid tests timeout
prop_generateAllResults fs xs ys =
  let fs' = map applyFun2 fs in
  take 10 (generateAllResults fs' xs ys) === take 10 [ f x y | f <- fs', x <- xs, y
```

Zippy lists

> There may be more than one way to make a parameterised type into an applicative functor. For example, the library `Control.Applicative` provides an alternative `zippy` instance for lists, in which the function `pure` makes an infinite list of copies of its argument, and the operator `<*>` applies each argument function to the corresponding argument value at the same position. Complete the given declarations that implement this idea.
>
> Note: The `ZipList` wrapper around the list type is required because each type can only have at most one instance declaration for a given class.

Solution:

```
newtype ZipList a = Z [a]
```

```haskell
  deriving (Show, Eq)

instance Functor ZipList where
  -- fmap :: (a -> b) -> ZipList a -> ZipList b
  fmap g (Z xs) = Z (map g xs)

instance Applicative ZipList where
  -- pure :: a -> ZipList a
  pure x = Z (repeat x)

  -- (<*>) :: ZipList (a -> b) -> ZipList a -> ZipList b
  (Z gs) <*> (Z xs) = Z (zipWith ($) gs xs)
```

Spec test:

```haskell
runZippy_spec :: ZipList a -> [a]
runZippy_spec (Z xs) = xs

fmap_type_test :: (a -> b) -> ZipList a -> ZipList b
fmap_type_test = fmap

prop_fmap_id :: [Int] -> Property
prop_fmap_id xs = runZippy_spec (fmap id (Z xs)) === xs

pure_type_test :: a -> ZipList a
pure_type_test = pure

prop_pure_repeat :: Int -> Property
prop_pure_repeat x = forAll (chooseInt (0,100)) $ \i -> runZippy_spec (pure x) !! i

ap_type_test :: ZipList (a -> b) -> ZipList a -> ZipList b
ap_type_test = (<*>)

prop_ap_id :: [Int] -> Property
prop_ap_id xs = runZippy_spec (pure id <*> (Z xs)) === xs

prop_id_ap :: [Int] -> Property
prop_id_ap xs = runZippy_spec (Z (map (flip ($)) xs) <*> (pure id)) === xs
```

Week 4A: Monads

Using Monads (1)

> A monad is an applicative functor that also supports the operations `return` and `(>>=)` (“bind”) with the following signature:

```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

One way to think about the bind operation is that `x >>= f` "extracts" a value (or several values) of type `a` from `x` and "feeds" it into the function `f`. The precise meaning of this depends on the concrete monad `m`. For example, the `Maybe` monad is implemented as follows:

```
instance Monad Maybe where
  return x       = Just x
  Nothing  >>= f = Nothing
  (Just x) >>= f = f x
```

For this exercise, you are given two functions `safeSquareRoot` and `multiplyIfSmall`. Using these two functions and the bind operation, implement the function `sqrtAndMultiply :: Double -> Maybe Double` that takes the square root of the input and then multiplies the result by 10 if it is small.

Note. Your solution should not make explicit use of the `Nothing` and `Just` constructors of the `Maybe` type, only the `safeSquareRoot` and `multiplyIfSmall` functions and the `(>>=)` operation. Try to make your solution fit on a single short line!

Solution:

```
prop_negativeInput (Negative x) = sqrtAndMultiply (x :: Double) === Nothing

prop_smallInput (Positive x) = sqrt x < 9.5 ==> sqrtAndMultiply x === Just (10 * sqr

prop_largeInput (Positive x) = sqrtAndMultiply (9.5*9.5 + x) === Nothing
```

Spec test:


```
prop_negativeInput (Negative x) = sqrtAndMultiply (x :: Double) === Nothing

prop_smallInput (Positive x) = sqrt x < 9.5 ==> sqrtAndMultiply x === Just (10 * sqr

prop_largeInput (Positive x) = sqrtAndMultiply (9.5*9.5 + x) === Nothing
```


Using Monads (2)

The bind operation for the list monad is implemented as follows:

```
-- (>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = concat (map f xs)
```

Now use this bind operation for the list monad to implement a function `addAndNegate :: [Int] -> [Int]` that adds `1`, `2`, and `3` to each element in the input list, and then for each element in the result also includes the negation of that input. For example:

```
addAndNegate [1,2] = [2, -2, 3, -3, 4, -4, 3, -3, 4, -4, 5, -5]
```

Note. Your solution should not make explicit use of list comprehensions or functions such as `map` and `concat`, only the `(>>=)` operation. Try to make your solution fit on a single short line!

Solution:

```
prop_addAndNegate_correct xs = addAndNegate xs === [ y | x <- xs , y <- [x+1,-(x+1),
```

Spec test:

```
prop_addAndNegate_correct xs = addAndNegate xs === [ y | x <- xs , y <- [x+1,-(x+1),
```

Using do-notation

Rather than using the `>>=` operator directly, Haskell provides a more convenient syntax for writing monadic code: `do` -notation. In the previous exercises, you implemented a number of functions using the applicative operator `<*>` or the monadic bind operator `>>=`. Now re-implement them all using `do` -notation instead.

Note. Your solution should not make explicit use of constructors such as `Nothing` or `Just`, list comprehensions, operations such as `<*>` or `>>=`, or any library functions other than the ones given in the Library code.

Solution:

```
sumOfSquareRoots :: Double -> Double -> Maybe Double
sumOfSquareRoots x y = do
  sqrtx <- safeSquareRoot x
  sqrty <- safeSquareRoot y
  return (sqrtx + sqrty)

generateAllResults :: [Int -> Int -> Int] -> [Int] -> [Int] -> [Int]
generateAllResults fs xs ys = do
```

```
  f <- fs
  x <- xs
  y <- ys
  return (f x y)

sqrtAndMultiply :: Double -> Maybe Double
sqrtAndMultiply x = do
  sqrtx <- safeSquareRoot x
  result <- multiplyIfSmall 10 sqrtx
  return result

{- Alternative shorter version (without 'return'):
sqrtAndMultiply :: Double -> Maybe Double
sqrtAndMultiply x = do
  sqrtx <- safeSquareRoot x
  multiplyIfSmall sqrtx
-}

addAndNegate :: [Int] -> [Int]
addAndNegate xs = do
  x <- xs
  y <- [x+1,x+2,x+3]
  z <- [y,-y]
  return z

{- Alternative shorter version (without 'return'):
addAndNegate :: [Int] -> [Int]
addAndNegate xs = do
  x <- xs
  y <- [x+1,x+2,x+3]
  [x,-x]
-}
```

Spec test:

```
prop_sumOfSquareRoots_neg1 :: Negative Double -> Double -> Property
prop_sumOfSquareRoots_neg1 (Negative x) y = sumOfSquareRoots x y === Nothing

prop_sumOfSquareRoots_neg2 :: Double -> Negative Double -> Property
prop_sumOfSquareRoots_neg2 x (Negative y) = sumOfSquareRoots x y === Nothing

prop_sumOfSquareRoots_pospos :: Positive Double -> Positive Double -> Property
prop_sumOfSquareRoots_pospos (Positive x) (Positive y) = sumOfSquareRoots x y === Ju

-- Testing with a single operation +
prop_generateSums xs ys = generateAllResults [(+)] xs ys === [ x+y | x <- xs, y <- y

-- Testing with an arbitrary subset of {+,-,*}
prop_generateSumsAndProducts xs ys =
  forAllShow (sublistOf [("+",(+)),("-",(-)),("*",(*))]) (show . map fst) $ \fs ->
```

```
  let fs' = map snd fs in
  generateAllResults fs' xs ys === [ f x y | f <- fs', x <- xs, y <- ys ]

-- Testing with a list of arbitrary functions generated by QuickCheck
-- Only look at the first 10 elements to avoid tests timeout
prop_generateAllResults fs xs ys =
  let fs' = map applyFun2 fs in
  take 10 (generateAllResults fs' xs ys) === take 10 [ f x y | f <- fs', x <- xs, y

prop_negativeInput (Negative x) = sqrtAndMultiply (x :: Double) === Nothing

prop_smallInput (Positive x) = sqrt x < 9.5 ==> sqrtAndMultiply x === Just (10 * sqr

prop_largeInput (Positive x) = sqrtAndMultiply (9.5*9.5 + x) === Nothing

prop_addAndNegate_correct xs = addAndNegate xs === [ y | x <- xs , y <- [x+1,-(x+1),
```

Using the Reader monad (1)

> So far we have seen how to use the `Maybe` and list monads. Another commonly used monad is the `Reader` monad, which represents a computation context where computations have access to a read-only shared global variable. The type `Reader r a` is parametrized by the type of the global variable `r`, and the return type of the computation `a`.
>
> The simplest function you can call within this monad is `ask`, which retrieves the value of the global variable:
>
> ```
> ask :: Reader r r
> ```
>
> For example, the following function retries the value of the global variable stored in the `Reader`, and returns a string of this value incremented by 2:
>
> ```
> add2AndShow :: Reader Int String
> add2AndShow = do
>   i <- ask
>   return (show (i + 2))
> ```
>
> Meanwhile, the `runReader` function takes as input a `Reader` computation and the value of the global variable, and returns the result of running the computation with that value.
>
> ```
> runReader :: Reader r a -> r -> a
> ```
>
> For example, `runReader add2AndShow 4` evaluates to the string `"6"`.

> Two other `Reader` functions you can use are `asks` and `local`.
>
> ```
> asks :: (r -> a) -> Reader r a
> local :: (r -> r) -> Reader r a -> Reader r a
> ```
>
> The `asks` function works like `ask`, except that it will apply the given function to the value of the global variable before returning it. For example, we could have written `add2AndShow` like this instead:
>
> ```
> add2AndShow :: Reader Int String
> add2AndShow = do
>   i <- asks (+2)
>   return (show i)
> ```
>
> Meanwhile, the `local` function will run the given `Reader` computation where the value of the global variable has been modified using the given function. As the name suggests this modification is only done locally, so subsequent computations will use the original state. For example:
>
> ```
> get3Values :: Reader Int (Int, Int, Int)
> get3Values = do
>   x <- ask
>   y <- local (+1) ask
>   z <- ask
>   return (x,y,z)
> ```
>
> Evaluating `runReader get3Values 5` will result in the tuple `(5,6,5)`.
>
> **Task.** Now implement a function `add2AndShowDouble` that works like `add2AndShow` but also shows twice the original value. For example, `runReader add2AndShowDouble 3` should return `"(5,6)"`.
>
> **Hint.** There are many different ways to implement this function. Try to find different ones that use all the operations `ask`, `asks`, and `local`.

Solution:

```
-- Option 1: using `ask`:
add2AndShowDouble :: Reader Int String
add2AndShowDouble = do
  x <- ask
  return (show (x+2,x*2))

-- Option 2: using `asks`:
add2AndShowDouble' :: Reader Int String
```

```haskell
add2AndShowDouble' = do
  x <- asks (+2)
  y <- asks (*2)
  return (show (x,y))

-- Option 3: using `local`:
add2AndShowDouble'' :: Reader Int String
add2AndShowDouble'' = do
  x <- local (+2) ask
  y <- local (*2) ask
  return (show (x,y))
```

Spec test:

```haskell
prop_add2AndShowDouble_correct x = runReader add2AndShowDouble x === show (x+2,x*2)
```

Using the Reader monad (2)

> In the previous exercise you learned how to use the `Reader` monad. As a reminder, here are the most important operations related to the `Reader` type:
>
> ```haskell
> ask :: Reader r r
> runReader :: Reader r a -> r -> a
> asks :: (r -> a) -> Reader r a
> local :: (r -> r) -> Reader r a -> Reader r a
> ```
>
> Now suppose you are implementing a program that manages user data using the following datatype:
>
> ```haskell
> data User = User
>   { userEmail :: String
>   , userPassword :: String
>   , userName :: String
>   , userAge :: Int
>   , userBio :: String
>   }
> ```
>
> Implement the following functions using the `Reader` monad:
>
> - The function `checkPassword :: String -> Reader User Bool` that checks whether the given password is equal to the user's password.
> - The function `displayProfile :: Reader User [String]` that displays the user's data in the following format:

> ```
> Name: {name}
> Age: {age}
> Bio: {bio}
> ```
>
> - The function `authAndDisplayProfile :: User -> String -> Maybe [String]` that returns the user's profile if the given password is correct, or `Nothing` otherwise.

Solution:

```haskell
data User = User
  { userEmail :: String
  , userPassword :: String
  , userName :: String
  , userAge :: Int
  , userBio :: String
  }

checkPassword :: String -> Reader User Bool
checkPassword givenPassword = do
  password <- asks userPassword
  return (password == givenPassword)

{- Alternative shorter version
checkPassword :: String -> Reader User Bool
checkPassword = asks . (==)
-}

displayProfile :: Reader User [String]
displayProfile = do
  name <- asks userName
  age <- asks userAge
  profile <- asks userBio
  return [ "Name: " ++ name , "Age: " ++ show age, "Bio: " ++ profile ]

authAndDisplayProfile :: User -> String -> Maybe [String]
authAndDisplayProfile user givenPassword =
  if runReader (checkPassword givenPassword) user
  then Just (runReader displayProfile user)
  else Nothing
```

Spec test:

```haskell
prop_checkPassword_correct mail pwd name age bio
  = runReader (checkPassword pwd) (User mail pwd name age bio) === True

prop_checkPassword_incorrect mail pwd name age bio pwd'
  = pwd /= pwd' ==> runReader (checkPassword pwd') (User mail pwd name age bio) ===
```

```
prop_display_correct mail pwd name age bio
  = runReader displayProfile (User mail pwd name age bio) === [ "Name: " ++ name , '

prop_authAndDisplay_correct mail pwd name age bio
  = authAndDisplayProfile (User mail pwd name age bio) pwd === Just (runReader displ

prop_authAndDisplay_incorrect mail pwd name age bio pwd'
  = pwd /= pwd' ==> authAndDisplayProfile (User mail pwd name age bio) pwd' === Noth
```

Implementing the Reader monad

> In a previous exercise you have used the `Reader` type which captures the effect of a global read-only variable. It is defined as follows:
>
> ```
> newtype Reader r a = Reader (r -> a)
> ```
>
> This definition has been given together with the functions `ask`, `asks`, `local`, and `runReader`.
>
> Your assignment is to complete the given instance declarations to make `Reader` into an instance of `Functor`, `Applicative`, and `Monad`.
>
> **Hint.** For implementing the `Monad` instance in particular, the helper function `runReader` (defined in the library code) may be useful.

```
newtype Reader r a = Reader (r -> a)

-- The ask function gets the value of the global variable stored
-- in the Reader.
ask :: Reader r r
ask = Reader id

-- The asks function gets the value of the global variable and
-- applies the given function to it.
asks :: (r -> a) -> Reader r a
asks f = Reader f

-- The local function allows running a Reader action with a
-- different value of the local variable.
local :: (r -> r) -> Reader r a -> Reader r a
local g (Reader f) = Reader (f . g)

-- The runReader function unwraps a Reader r a value and returns
-- it as a function from r to a.
runReader :: Reader r a -> r -> a
runReader (Reader f) = f
```

```haskell
instance Functor (Reader r) where
  -- fmap :: (a -> b) -> Reader r a -> Reader r b
  fmap f (Reader g) = Reader (f . g)

instance Applicative (Reader r) where
  -- pure :: a -> Reader r a
  pure x = Reader (const x)

  -- (<*>) :: Reader r (a -> b) -> Reader r a -> Reader r b
  (Reader f) <*> (Reader g) = Reader (\x -> f x (g x))

instance Monad (Reader r) where
  -- return :: a -> Reader r a
  return = pure

  -- (>>=) :: Reader r a -> (a -> Reader r b) -> Reader r b
  Reader f >>= g = Reader (\x -> runReader (g (f x)) x)
```

Spec test:

```haskell
-- An example of using the Reader monad
reader_example :: Reader Int (Int,Int)
reader_example = do
  x <- asks (*5)         -- get current value of global variable multiplied by 5
  y <- asks (+3)         -- get current value of global variable plus 3
  z <- local (+1) $ do   -- locally add 1 to the global variable
    a <- asks (*5)
    b <- asks (+3)
    return (a+b)
  return (x+y, z)

prop_reader_example :: Property
prop_reader_example = runReader reader_example 1 === (9,15)

runReader_spec :: Reader r a -> r -> a
runReader_spec (Reader f) x = f x

fmap_type_test :: (a -> b) -> Reader r a -> Reader r b
fmap_type_test = fmap

prop_fmap_id :: Fun Int Int -> Int -> Property
prop_fmap_id (Fun _ f) x = runReader_spec (fmap id (Reader f)) x === f x

prop_id_fmap :: Fun Int Int -> Int -> Property
prop_id_fmap (Fun _ f) x = runReader_spec (fmap f (Reader id)) x === f x

pure_type_test :: a -> Reader r a
pure_type_test = pure
```

```haskell
prop_pure_const :: Int -> Int -> Property
prop_pure_const x y = runReader_spec (pure x) y === x

ap_type_test :: Reader r (a -> b) -> Reader r a -> Reader r b
ap_type_test = (<*>)

prop_ap_id :: Fun Int Int -> Int -> Property
prop_ap_id (Fun _ f) x = runReader_spec (pure id <*> (Reader f)) x === f x

return_type_test :: a -> Reader r a
return_type_test = return

bind_type_test :: Reader r a -> (a -> Reader r b) -> Reader r b
bind_type_test = (>>=)

prop_return_const :: Int -> Int -> Property
prop_return_const x y = runReader_spec (return x) y === x

prop_bind_return :: Fun Int Int -> Int -> Property
prop_bind_return (Fun _ f) x = runReader_spec (Reader f >>= \x -> return x) x === f
```

Using the Writer monad

> While the `Reader` monad gave us access to a global variable that is read-only, the `Writer` monad gives us one that is *write-only*, which can only be accessed when our computation is complete. The key to this is that the type of the global variable should be a `Monoid`, so that values that are written can be combined using `<>`.
>
> ```haskell
> instance (Monoid w) => Monad (Writer w) where
>   ...
> ```
>
> The primary function for using the `Writer` monad is `tell`, which takes a value of type `w` and appends it to the current state:
>
> ```haskell
> tell :: w -> Writer w ()
> ```
>
> Once we have a complete computation, we can extract the result together with the final value of the global variable using `runWriter`:
>
> ```haskell
> runWriter :: Writer w a -> (a, w)
> ```
>
> Unlike the `Reader` monad, we do not need to provide an initial value for the global variable, instead `runWriter` uses `mempty` as the default value.

> In this exercise, we will use `Monoid` for implementing a simple calculator in a way that also allows us to keep track of the total "cost" of the operations and to generate a detailed log of all operations that were applied.
>
> To start with, we define a datatype of the operations supported by our calculator:
>
> ```
> data Op = Add Double | Subtract Double | Multiply Double | Divide Double | Sqrt
> ```
>
> In the code, there are also two functions `opCost` and `opLog` for calculating the cost of each operation and for writing log messages, respectively.
>
> **Tasks.**
>
> 1. Implement the function `applyOpCount` that applies an operation to the given input and also writes the cost of the operation to the state stored in the `Writer`. Then use this function to apply a series of operations to an input value in the `applyAndCountOperations` function.
> 2. Implement the function `applyOpLog` that applies an operation to the given input and also writes a log message to the state stored in the `Writer`. Then use this function to apply a series of operations to an input value in the `applyAndLogOperations` function.
>
> See the 'Test' tab for some examples of how these functions should work. When taking the square root of a negative number, you can assume the output value is unchanged.

Solution:

```haskell
import Data.Monoid

data Op = Add Double | Subtract Double | Multiply Double | Divide Double | Sqrt
  deriving (Show)

opCost :: Op -> Sum Int
opCost (Add _) = Sum 1
opCost (Subtract _) = Sum 2
opCost (Multiply _) = Sum 5
opCost (Divide _) = Sum 10
opCost Sqrt = Sum 20

opLog (Add x) = "Adding " ++ show x
opLog (Subtract x) = "Subtracting " ++ show x
opLog (Multiply x) = "Multiplying by " ++ show x
opLog (Divide x) = "Dividing by " ++ show x
opLog Sqrt = "Taking Square Root"

applyOp :: Op -> Double -> Double
applyOp (Add x) y = x + y
applyOp (Subtract x) y = y - x
```

```haskell
applyOp (Multiply x) y = x * y
applyOp (Divide x) y = y / x
applyOp Sqrt y = if y >= 0 then sqrt y else y

applyOpCount :: Op -> Double -> Writer (Sum Int) Double
applyOpCount op y = do
  tell (opCost op)
  return (applyOp op y)

applyOpsCount :: [Op] -> Double -> Writer (Sum Int) Double
applyOpsCount [] x = return x
applyOpsCount (op:ops) x = do
  y <- applyOpCount op x
  applyOpsCount ops y

applyAndCountOperations :: [Op] -> Double -> (Double, Sum Int)
applyAndCountOperations ops y = runWriter (applyOpsCount ops y)

applyOpLog :: Op -> Double -> Writer [String] Double
applyOpLog op y = do
  tell [opLog op]
  return (applyOp op y)

applyOpsLog :: [Op] -> Double -> Writer [String] Double
applyOpsLog [] x = return x
applyOpsLog (op:ops) x = do
  y <- applyOpLog op x
  applyOpsLog ops y

applyAndLogOperations :: [Op] -> Double -> (Double, [String])
applyAndLogOperations ops y = runWriter (applyOpsLog ops y)
```

Spec test:

```haskell
import Data.Monoid

instance Arbitrary Op where
  arbitrary = oneof [Add <$> arbitrary, Subtract <$> arbitrary, Multiply <$> arbitra

prop_applyOpCount_add x y = runWriter (applyOpCount (Add x) y) === (x+y, Sum 1)
prop_applyOpCount_sub x y = runWriter (applyOpCount (Subtract x) y) === (y-x, Sum 2)
prop_applyOpCount_mult x y = runWriter (applyOpCount (Multiply x) y) === (x*y, Sum 5
prop_applyOpCount_div (NonZero x) y = runWriter (applyOpCount (Divide x) y) === (y/x
prop_applyOpCount_sqrt (Positive x) = runWriter (applyOpCount Sqrt x) === (sqrt x, S

prop_applyOpLog_add x y = runWriter (applyOpLog (Add x) y) === (x+y, ["Adding " ++ s
prop_applyOpLog_sub x y = runWriter (applyOpLog (Subtract x) y) === (y-x, ["Subtract
prop_applyOpLog_mult x y = runWriter (applyOpLog (Multiply x) y) === (x*y, ["Multipl
prop_applyOpLog_div (NonZero x) y = runWriter (applyOpLog (Divide x) y) === (y/x, ["
prop_applyOpLog_sqrt (Positive x) = runWriter (applyOpLog Sqrt x) === (sqrt x, ["Tak
```

```haskell
applyOp_spec :: Op -> Double -> Double
applyOp_spec (Add x) y = x + y
applyOp_spec (Subtract x) y = y - x
applyOp_spec (Multiply x) y = x * y
applyOp_spec (Divide x) y = y / x
applyOp_spec Sqrt y = if y >= 0 then sqrt y else y

prop_applyAndCount_empty x = applyAndCountOperations [] x === (x, Sum 0)
prop_applyAndLog_empty x = applyAndLogOperations [] x === (x, [])

prop_applyAndCount_append op ops x = applyAndCountOperations (op:ops) x === (z , opC
  where
    (z, cost) = applyAndCountOperations ops (applyOp_spec op x)

prop_applyAndLog_append op ops x = applyAndLogOperations (op:ops) x === (z , [opLog
  where
    (z, log) = applyAndLogOperations ops (applyOp_spec op x)
```

Implementing the Writer monad

> The `Writer` type and the functions `tell` and `runWriter` that you used in the previous exercises are defined as follows:
>
> ```haskell
> newtype Writer w a = Writer (a, w)
>
> tell :: w -> Writer w ()
> tell x = Writer ((), x)
>
> runWriter :: Writer w a -> (a, w)
> runWriter (Writer x) = x
> ```
>
> Now, define instances of the `Functor`, `Applicative`, and `Monad` classes for the `Writer w` type.
>
> **Hint.** For implementing the `Monad` instance, the helper function `runWriter :: Writer w a -> (a, w)` may be useful.

Solution:

```haskell
newtype Writer w a = Writer (a, w)

tell :: w -> Writer w ()
tell x = Writer ((), x)

runWriter :: Writer w a -> (a, w)
runWriter (Writer x) = x
```

```haskell
instance Functor (Writer w) where
  -- fmap :: (a -> b) -> Writer w a -> Writer w b
  fmap f (Writer (x, w)) = Writer (f x, w)

instance Monoid w => Applicative (Writer w) where
  -- pure :: a -> Writer w a
  pure x = Writer (x, mempty)

  -- (<*>) :: Writer w (a -> b) -> Writer w a -> Writer w b
  Writer (f, w1) <*> Writer (x, w2) = Writer (f x, w1 <> w2)

instance Monoid w => Monad (Writer w) where
  -- return :: a -> Writer w a
  return   = pure

  -- (>>=) :: Writer w a -> (a -> Writer w b) -> Writer w b
  Writer (x, w1) >>= f =
    let (y, w2) = runWriter (f x)
    in  Writer (y, w1 <> w2)
```

Spec test:

```haskell
multWithLog :: Int -> Int -> Writer [String] Int
multWithLog x y = do
    tell ["Multiplying " ++ show x ++ " and " ++ show y]
    return (x*y)

prop_multWithLog_example :: Property
prop_multWithLog_example = runWriter act === (30, ["Multiplying 3 and 5", "Multiplyi
  where
    act = do
      x <- multWithLog 3 5
      y <- multWithLog x 2
      return y

fmap_type_test :: (a -> b) -> Writer w a -> Writer w b
fmap_type_test = fmap

prop_fmap_id :: [Int] -> Int -> Property
prop_fmap_id w x = runWriter (fmap id (Writer (x, w))) === (x, w)

prop_fmap_empty :: Fun Int Int -> Int -> Property
prop_fmap_empty (Fun _ f) x = runWriter (fmap f (Writer (x, []))) === (f x, ([] :: [

pure_type_test :: Monoid w => a -> Writer w a
pure_type_test = pure

prop_pure_empty :: Int -> Property
prop_pure_empty x = runWriter (pure x) === (x, ([] :: [Int]))
```

```haskell
ap_type_test :: Monoid w => Writer w (a -> b) -> Writer w a -> Writer w b
ap_type_test = (<*>)

prop_ap_id :: [Int] -> Int -> Property
prop_ap_id w x = runWriter (pure id <*> (Writer (x, w))) === (x, w)

return_type_test :: Monoid w => a -> Writer w a
return_type_test = return

bind_type_test :: Monoid w => Writer w a -> (a -> Writer w b) -> Writer w b
bind_type_test = (>>=)

prop_return_empty :: Int -> Property
prop_return_empty x = runWriter (return x) === (x, ([] :: [Int]))

prop_bind_return :: [Int] -> Int -> Property
prop_bind_return w x = runWriter (Writer (x, w) >>= return) === (x, w)
```

Using the State monad

> The *State* monad combines the functionality of the Reader and Writer monads. We have a single stateful object, and we are free to access and read from it, or update and change its values. When we change the object, subsequent operations in the monad will refer to the updated value. Note the state does NOT have to be a Monoid, as with Writer.
>
> The two most important operations of the `State` Monad are `get` (which retrieves the current state), `put` (which replaces the current state with a new value), and `runState` (which runs the computation given an initial value of the state, and returns both the result and the final value of the state):
>
> ```haskell
> get :: State s s
> put :: s -> State s ()
> runState :: State s a -> s -> (a, s)
> ```
>
> If you only care about the final computation result, you can use `evalState` instead of `runState`. If you only care about the final state, you can use `execState`:
>
> ```haskell
> evalState :: State s a -> s -> a
> execState :: State s a -> s -> s
> ```
>
> There are two other functions you can use. Just like we have `asks` in Reader, there is `gets` which can retrieve a field from the State.

```haskell
gets :: (s -> a) -> State s a
```

Then you can use `modify` to apply a function on the state:

```haskell
modify :: (s -> s) -> State s ()
```

For example, `execState (modify (+4)) 5` evaluates to `9` .

Assignment. Use the `State` monad to implement a function `counter :: [Char] -> State (Int, Bool) Int` that takes as input a list of characters and interprets each character as follows: - `'a'` should increase the counter by `1` - `'b'` should decrease the counter by `1` - `'c'` should toggle the counter off, ignoring any `'a'` and `'b'` values until another `'c'` is encountered. The function `counter` uses a state of type `(Int, Bool)` , where the first component indicates the current value of the counter, and the second component indicates whether the counter is currently on or off.

You can find some examples in the "Test" tab.

Solution:

```haskell
-- Increase the counter by 1
increaseCounter :: State (Int, Bool) ()
increaseCounter = modify (\(c,b) -> (c+1,b))

-- Decrease the counter by 1
decreaseCounter :: State (Int, Bool) ()
decreaseCounter = modify (\(c,b) -> (c-1,b))

-- Toggle the boolean flag from True to False or vice versa
toggleFlag :: State (Int, Bool) ()
toggleFlag = modify (\(c,b) -> (c,not b))

-- Do nothing
doNothing :: State (Int, Bool) ()
doNothing = return ()

-- Execute an action only when the boolean flag is true,
-- and do nothing otherwise.
whenFlagOn :: State (Int, Bool) () -> State (Int, Bool) ()
whenFlagOn action = do
  b <- gets snd
  if b then action else doNothing

counter :: [Char] -> State (Int, Bool) Int
counter [] = gets fst
counter (c:cs) = do
```

```
    case c of
      'a' -> whenFlagOn increaseCounter
      'b' -> whenFlagOn decreaseCounter
      'c' -> toggleFlag
      _   -> doNothing
    counter cs
```

Spec test:

```
prop_counter_empty_eval s = evalState (counter "") s === fst s
prop_counter_empty_exec s = execState (counter "") s === s
prop_counter_a_true n = runState (counter "a") (n, True) === (n+1, (n+1, True))
prop_counter_a_false n = runState (counter "a") (n, False) === (n, (n, False))
prop_counter_b_true n = runState (counter "b") (n, True) === (n-1, (n-1, True))
prop_counter_b_false n = runState (counter "b") (n, False) === (n, (n, False))
prop_counter_c b n = runState (counter "c") (n, b) === (n, (n, not b))

prop_counter_others xs s = runState (counter xs) s === runState (counter xs') s
  where xs' = filter (\x -> x == 'a' || x == 'b' || x == 'c') xs

prop_counter_cons s =
  forAll (elements "abc") $ \x ->
  forAll (listOf (elements "abc")) $ \xs ->
  runState (counter (x:xs)) s === runState (counter xs) (execState (counter [x]) s)
```

Sequencing data

> One big advantage of having a general concept of monads is that it is possible to write generic code that works in *any* monad. One example of this is the function `sequence :: Monad m => [m a] -> m [a]` that takes a list of monadic actions, and evaluates them in left-to-right sequence, collecting all the results into a list. The goal of this exercise is to implement this library function yourself.

Solution:

```
import Prelude hiding (sequence)

sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (m:ms) = do
  x <- m
  xs <- sequence ms
  return (x:xs)
```

Spec test:

```haskell
import Prelude hiding (sequence)
import Data.Functor.Identity

prop_sequence_example1 :: Property
prop_sequence_example1 = sequence [Just 1, Just 2, Just 3] === Just [1,2,3]

prop_sequence_example2 :: Property
prop_sequence_example2 = sequence [Left "oops!", Right 42, Left "oh no..."] === Left

sequence_spec :: Monad m => [m a] -> m [a]
sequence_spec [] = return []
sequence_spec (m:ms) = do
  x <- m
  xs <- sequence_spec ms
  return (x:xs)

prop_sequence_identity :: [Identity Int] -> Property
prop_sequence_identity xs = sequence xs === sequence_spec xs

prop_sequence_either :: [Either Int Int] -> Property
prop_sequence_either xs = sequence xs === sequence_spec xs

prop_sequence_reader :: [Fun Int Int] -> Int -> Property
prop_sequence_reader fs x = sequence (map applyFun fs) x === sequence_spec (map appl
```

Monadic Filter

> Reimplement the library function `filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]` , that takes a (monadic) predicate `a -> m Bool` and uses this to filter a given list.
>
> **Note.** `filterM` must process the list elements left-to-right, and it must preserve the order of the elements of the input list, as far as they appear in the result.

Solution:

```haskell
import Prelude hiding (filterM)

filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p [] = return []
filterM p (x:xs) = do
  keep <- p x
  ys   <- filterM p xs
  if keep then return (x:ys) else return ys
```

Spec test:

```haskell
import Prelude hiding (filterM)
import Data.Functor.Identity

-- Keeping all the divisors of a given number. If any division by 0 happens, the who
prop_filterM_divisors :: Property
prop_filterM_divisors = filterM (isDivisorOf 10) [1..10] === Just [1,2,5,10]
  where
    isDivisorOf x y | y == 0    = Nothing
                    | otherwise = Just (x `mod` y == 0)

prop_filterM_divisors_error :: Property
prop_filterM_divisors_error = filterM (isDivisorOf 10) [0..10] === Nothing
  where
    isDivisorOf x y | y == 0    = Nothing
                    | otherwise = Just (x `mod` y == 0)

filterM_spec :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM_spec p [] = return []
filterM_spec p (x:xs) = do
  keep <- p x
  ys   <- filterM_spec p xs
  if keep then return (x:ys) else return ys

prop_filterM_identity :: Fun Int (Identity Bool) -> [Int] -> Property
prop_filterM_identity (Fun _ f) xs = filterM f xs === filterM_spec f xs

prop_filterM_either :: Fun Int (Either Int Bool) -> [Int] -> Property
prop_filterM_either (Fun _ f) xs = filterM f xs === filterM_spec f xs

prop_filterM_reader :: Fun Int (Fun Int Bool) -> [Int] -> Int -> Property
prop_filterM_reader (Fun _ f) xs x = filterM (applyFun . f) xs x === filterM_spec (a
```

A functional while loop

> Some algorithms are expressed more naturally as an imperative `while` loop instead of as a recursive function. Implement a monadic function `while :: Monad m => (m Bool) -> m () -> m ()` that takes as arguments a loop condition `cond`, and a loop body `body`, and repeatedly runs the loop body as long as the condition returns `True`.
>
> As an example of how this function `while` might be used, the test template contains an implementation of Euclid's algorithm `euclid :: Int -> Int -> Int` for finding the greatest common divisor of two positive numbers, using the `State` monad with a state of type `(Int,Int)`.

Solution:

```haskell
while :: Monad m => (m Bool) -> m () -> m ()
```

```haskell
while loopCond loopBody = do
  continue <- loopCond
  if continue then do
    loopBody
    while loopCond loopBody
  else
    return ()
```

Spec test:

```haskell
euclid :: Int -> Int -> Int
euclid x y = fst (execState euclidLoop (x,y))
  where
    euclidLoop = while (do (x,y) <- getState; return (x /= y)) (do
        (x,y) <- getState
        if x < y then putState (x,y-x) else putState (x-y,y)
      )

prop_euclid_correct :: Positive Int -> Positive Int -> Property
prop_euclid_correct (Positive x) (Positive y) = euclid x y === gcd x y
```

Expr monad

> Consider the following type `Expr a` of arithmetic expressions that contain variables of some type `a` :
>
> ```haskell
> data Expr a = Var a | Val Int | Add (Expr a) (Expr a)
>   deriving (Show)
> ```
>
> For example, if we want to represent variables as string we can use the type `Expr String` . The library code defines this datatype and an instance of the `Functor` typeclass. Show how to make this type into an instance of the classes `Applicative` and `Monad` .
>
> **Hint.** It may be easier to implement the `Monad` instance first and derive the implementation of `Applicative` after. Intuitively, the behaviour of `e >>= f` is to replace each variable in the expression `e` with a new expression, which is produced by applying the function `f` to the variable. For example:
>
> ```haskell
> > let f "x" = Val 1; f "y" = Add (Val 1) (Var "z")
> > Add (Var "x") (Var "y") >>= f
> Add (Val 1) (Add (Val 1) (Var "z"))
> ```

Solution:

```haskell
instance Applicative Expr where
  -- pure :: a -> Expr a
  pure = Var

  -- (<*>) :: Expr (a -> b) -> Expr a -> Expr b
  Var f <*> xe = fmap f xe
  Val i <*> xe = Val i
  Add fe ge <*> xe = Add (fe <*> xe) (ge <*> xe)


instance Monad Expr where
  -- return :: a -> Expr a
  return = pure

  -- (>>=) :: Expr a -> (a -> Expr b) -> Expr b
  Var x >>= f = f x
  Val i >>= f = Val i
  (Add u v) >>= f = Add (u >>= f) (v >>= f)
```

Spec test:

```haskell
prop_example' :: Property
prop_example' = (Add (Var "x") (Var "y") >>= f) === (Add (Val 1) (Add (Val 1) (Var '
  where
    f "x" = Val 1
    f "y" = Add (Val 1) (Var "z")

prop_bind_var' :: Property
prop_bind_var' = (Var "x" >>= \_ -> Add (Var "y") (Var "z")) === Add (Var "y") (Var

instance Arbitrary a => Arbitrary (Expr a) where
  arbitrary = sized expr'
    where
      expr' 0       = oneof [ Var <$> arbitrary, Val <$> arbitrary ]
      expr' n | n>0 = oneof [ Var <$> arbitrary
                            , Val <$> arbitrary
                            , Add <$> expr' m <*> expr' m
                            ]
        where m = n `div` 2

  shrink (Var x) = map Var $ shrink x
  shrink (Val x) = map Val $ shrink x
  shrink (Add x y) = [x,y] ++ [Add x y' | y' <- shrink y ] ++ [ Add x' y | x' <- shr

prop_pure_var :: Int -> Property
prop_pure_var x = pure x === Var x

prop_return_var :: Int -> Property
prop_return_var x = return x === Var x
```

```
prop_bind_var :: Int -> (Fun Int (Expr Int)) -> Property
prop_bind_var x (Fun _ f) = (Var x >>= f) === f x

prop_bind_val :: Int -> (Fun Int (Expr Int)) -> Property
prop_bind_val x (Fun _ f) = (Val x >>= f) === Val x

prop_bind_return :: Expr Int -> Property
prop_bind_return e = (e >>= return) === e

prop_bind_assoc :: Expr Int -> (Fun Int (Expr Int)) -> (Fun Int (Expr Int)) -> Prope
prop_bind_assoc x (Fun _ f) (Fun _ g) = ((x >>= f) >>= g) === (x >>= (\y -> f y >>=

prop_ap_correct :: Expr (Fun Int Int) -> Expr Int -> Property
prop_ap_correct fe xe = (fmap applyFun fe <*> xe) === (fmap applyFun fe `ap` xe)
  where
    ap m1 m2 = do
      x1 <- m1
      x2 <- m2
      return (x1 x2)
```

Week 4B: Lazy Evaluation

Fibonacci

> Using a list comprehension, define an expression `fibs :: [Integer]` that generates the infinite list of Fibonacci numbers
>
> $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$
>
> using the following simple procedure:
>
> - the first two numbers are 0 and 1;
> - the next is the sum of the previous two;
> - return to the second step.
>
> **Hint:** make use of the library functions `zip` and `tail`. Note that numbers in the Fibonacci sequence quickly become large, hence the use of the type `Integer` of arbitrary-precision integers above.

Solution:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Spec test:

```
prop_fib_first :: Property
prop_fib_first = head fibs === 0

prop_fib_second :: Property
prop_fib_second = head (tail fibs) === 1

prop_fib_next :: Property
prop_fib_next = forAll (chooseInt (0,10000)) $ \i ->
  within 1000000 $
  fibs !! (i+2) === fibs !! (i+1) + fibs !! i
```

Newton's Method

> *Newton's method* for computing the square root of a (non-negative) floating-point number `n` can be expressed as follows:
>
> - start with an initial approximation to the result;
> - given the current approximation `a`, the next approximation is defined by the function `next a = (a + n/a) / 2`
> - repeat the second step until the two most recent approximations are within some desired distance of one another, at which point the most recent value is returned as the result.
>
> Define a function `sqroot :: Double -> Double` that implements this procedure.
>
> **Hint:** first produce an infinite list of approximations using the library function `iterate`. For simplicity, take the number `1.0` as the initial approximation, and `0.00001` as the distance value.

Solution:

```
sqroot :: Double -> Double
sqroot n = loop approximations
  where
    next a = (a+n/a)/2
    approximations = iterate next 1.0
    loop (x:y:ys)
      | abs (x-y) < 0.00001 = y
      | otherwise           = loop (y:ys)
```

Spec test:

```
prop_sqroot_correct :: NonNegative Double -> Property
```

```
prop_sqroot_correct (NonNegative x) = within 1000000 $ abs (sqroot x - sqrt x) < 0.0
```

Prime numbers

> Write a function `primes :: [Integer]` that returns the infinite list of all prime numbers.
>
> **Hint.** First implement a function `sieve :: [Integer] -> [Integer]` that uses the Sieve of Eratosthenes to filter out any elements that are a multiple of a previous element, and then apply this function to the infinite list `[2..]`.

Solution:

```
sieve :: [Integer] -> [Integer]
sieve (x:xs) = x:sieve (filter (\y -> y `mod` x /= 0) xs)

primes = sieve [2..]
```

Spec test:

```
primes_spec n = take n $ sieve [2..]
  where
    sieve :: [Integer] -> [Integer]
    sieve []     = []
    sieve (x:xs) = x:sieve (filter (\y -> y `mod` x /= 0) xs)

prop_primes_prime :: NonNegative Int -> Property
prop_primes_prime (NonNegative i) = is_prime (primes !! i)
  where
    is_prime :: Integer -> Property
    is_prime n = n === 2 .||. n === 3 .||. forAll (chooseInteger (2,n-1)) (\i -> n `

prop_primes_first_hundred :: Property
prop_primes_first_hundred = take 100 primes === primes_spec 100
```

Cutting off branches

> Consider the following type of trees with values stored in the nodes:
>
> ```
> data Tree a = Node (Tree a) a (Tree a)
>             | Leaf
>   deriving (Show, Eq)
> ```
>
> Because of lazy evaluation in Haskell, it is possible to construct infinite trees of this type, for

> example:
>
> ```
> infiniteTree :: Int -> Tree Int
> infiniteTree n = Node (infiniteTree (n+1)) n (infiniteTree (n+1))
> ```
>
> This function constructs an infinite tree where the root has label `n`, the layer beneath that has label `n+1`, the layer beneath that has label `n+2`, etc.
>
> Implement a function `cutoff :: Int -> Tree a -> Tree a` that cuts off all branches of the tree beyond the given depth, by replacing them with `Leaf`. For example:
>
> ```
> > cutoff 0 (infiniteTree 0)
> Leaf
> > cutoff 1 (infiniteTree 0)
> Node Leaf 0 Leaf
> > cutoff 2 (infiniteTree 0)
> Node (Node Leaf 1 Leaf) 0 (Node Leaf 1 Leaf)
> > cutoff 3 (infiniteTree 0)
> Node (Node (Node Leaf 2 Leaf) 1 (Node Leaf 2 Leaf)) 0 (Node (Node Leaf 2 Leaf) 1
> ```

Solution:

```
cutoff :: Int -> Tree a -> Tree a
cutoff 0 _ = Leaf
cutoff n Leaf = Leaf
cutoff n (Node l x r) | n > 0 = Node (cutoff (n-1) l) x (cutoff (n-1) r)
```

Spec test:

```
prop_cutoff_leaf :: NonNegative Int -> Property
prop_cutoff_leaf (NonNegative c) = cutoff c (Leaf :: Tree Int) === Leaf

treeFromList :: [a] -> Gen (Tree a)
treeFromList [] = return Leaf
treeFromList (x:xs) = do
  i <- chooseInt (0,length xs)
  let (ys,zs) = splitAt i xs
  Node <$> treeFromList ys <*> pure x <*> treeFromList zs

prop_cutoff_zero :: [Int] -> Property
prop_cutoff_zero xs =
  forAll (treeFromList xs) $ \t ->
  cutoff 0 t === Leaf

prop_cutoff_node :: Positive Int -> Int -> [Int] -> [Int] -> Property
prop_cutoff_node (Positive c) x ys zs =
```

```haskell
  forAll (treeFromList ys) $ \l ->
  forAll (treeFromList zs) $ \r ->
  isNodeWith x (cutoff c (Node l x r))

  where isNodeWith x Leaf = False
        isNodeWith x (Node _ y _) = x == y

depth_spec :: Tree a -> Int
depth_spec Leaf = 0
depth_spec (Node l _ r) = 1 + max (depth_spec l) (depth_spec r)

prop_cutoff_depth :: Property
prop_cutoff_depth = within 1000000 $
  forAll (chooseInt (0,10)) $ \c ->
  depth_spec (cutoff c (infiniteTree 0)) === c
```

Flattening an infinite tree

> Consider the following type of trees with values stored in the nodes:
>
> ```haskell
> data Tree a = Node (Tree a) a (Tree a)
>             | Leaf
>   deriving (Show, Eq)
> ```
>
> Because of lazy evaluation in Haskell, it is possible to construct infinite trees of this type, for example:
>
> ```haskell
> infiniteTree :: Int -> Tree Int
> infiniteTree n = Node (infiniteTree (n+1)) n (infiniteTree (n+1))
> ```
>
> This function constructs an infinite tree where the root has label `n`, the layer beneath that has label `n+1`, the layer beneath that has label `n+2`, etc.
>
> Now implement a function `flatten :: Tree a -> [a]` that transforms a tree into a list of the labels in the tree, such that each label of an infinite tree occurs at a finite position in the list.
>
> **Note.** A simple depth-first traversal of the three will not work because it can get stuck on the left subtree of an infinite tree without ever getting to the right subtree!

Solution:

```haskell
interleave :: [a] -> [a] -> [a]
interleave []     ys     = ys
interleave xs     []     = xs
```

```haskell
interleave (x:xs) (y:ys) = x : y : interleave xs ys

flatten :: Tree a -> [a]
flatten Leaf = []
flatten (Node l x r) = x : interleave (flatten l) (flatten r)
```

Spec test:

```haskell
import Data.Set (Set)
import qualified Data.Set as Set

treeFromList :: [a] -> Gen (Tree a)
treeFromList [] = return Leaf
treeFromList (x:xs) = do
  i <- chooseInt (0,length xs)
  let (ys,zs) = splitAt i xs
  Node <$> treeFromList ys <*> pure x <*> treeFromList zs

prop_flatten_leaf :: Property
prop_flatten_leaf = flatten (Leaf :: Tree Int) === []

prop_flatten_single :: Int -> Property
prop_flatten_single x = flatten (Node Leaf x Leaf) === [x]

prop_flatten_finite :: [Int] -> Property
prop_flatten_finite xs = forAll (treeFromList xs) $ \t ->
  Set.fromList (flatten t) === Set.fromList xs

treeFromInfiniteList :: [a] -> Tree a
treeFromInfiniteList (x:xs) =
    let (ys,zs) = unterleave xs
    in  Node (treeFromInfiniteList ys) x (treeFromInfiniteList zs)
  where
    unterleave (y:z:xs) =
      let (ys,zs) = unterleave xs
      in  (y:ys,z:zs)

prop_flatten_infinite :: Property
prop_flatten_infinite = within 1000000 $
  let t = treeFromInfiniteList [0..] in
  forAll (chooseInt (0,20)) $ \i ->
  i `elem` flatten t
```

Evaluating factorial (call-by-name vs call-by-value)

> Consider two definitions of the function `fac :: Int -> Int` :
>
> ```haskell
> fac 0 = 1
> ```

```haskell
fac n = n * fac (n - 1)

fac' n = accum 1 n
  where
  accum x 0 = x
  accum x y = accum (x*y) (y-1)
```

1. Write down the evaluation sequences for `fac 3` under call-by-value reduction and call-by-name reduction. If there are multiple valid redexes to choose, pick the leftmost one first. Can you see a difference in the performance between the two strategies in the number of evaluation steps or the size of the intermediate expressions?
2. Write down the evaluation sequences for `fac' 3` under call-by-value reduction and call-by-name reduction. If there are multiple valid redexes to choose, pick the leftmost one first. Can you see a difference in the performance between the two strategies in the number of evaluation steps or the size of the intermediate expressions?
3. How would you modify the definition of `fac'` to improve its performance under the lazy evaluation strategy of Haskell?

Note. When writing down the evaluation sequences, you do not need to write intermediate steps for evaluation of functions from the Haskell prelude (such as `(-)` and `(*)`).

Call-by-value reduction of `fac 3`:

```
fac 3 --> 3 * fac 2
      --> 3 * (2 * fac 1)
      --> 3 * (2 * (1 * fac 0)))
      --> 3 * (2 * (1 * 1))) = 6
```

Call-by-name reduction of `fac 3`:

```
fac 3 --> 3 * fac 2
      --> 3 * (2 * fac 1)
      --> 3 * (2 * (1 * fac 0)))
      --> 3 * (2 * (1 * 1))) = 6
```

For `fac`, the choice of evaluation strategy does not matter.

Call-by-value reduction of `fac' 3`:

```
fac' 3 --> accum 1 3
       --> accum 3 2
       --> accum 6 1
       --> accum 6 0
       --> 6
```

Call-by-name reduction of `fac' 3`:

```
fac' 3 --> accum 1 3
       --> accum (1*3) 2
       --> accum ((1*3)*2) 1
       --> accum (((1*3)*2)*1) 0
       --> ((1*3)*2)*1 = 6
```

For `fac'` the number of evaluation steps is still the same under both strategies, but the size of intermediate expressions is much smaller under call-by-value.

You can use the strict application operator `($!)` to make `accum` strict in its first argument:

```
  fac' n = accum 1 n
    where
      accum x 0 = x
      accum x y = (accum $! (x*y)) (y-1)
```

Evaluating insertion sort

> Consider the following implementation of insertion sort in Haskell:
>
> ```
> ins :: Ord a => a -> [a] -> [a]
> ins x [] = x : []
> ins x (y:ys) | x <= y    = x : y : ys
>              | otherwise = y : (ins x ys)
>
> isort :: Ord a => [a] -> [a]
> isort []     = []
> isort (x:xs) = ins x (isort xs)
> ```
>
> 1. Write down the evaluation sequences for `head (isort [3,2,1])` under call-by-value reduction and call-by-name reduction. If there are multiple valid redexes to choose, pick the leftmost one first. Can you see a difference in the performance between the two strategies in the number of evaluation steps or the size of the intermediate expressions?
> 2. Consider now the expression `head (isort [n,n-1..1])` for some integer `n` (greater than 1). How many comparisons of two numbers are performed during the call-by-value and call-by-name reduction of this expression? What can we conclude about the complexity of evaluating this expression?
> 3. Now suppose we instead want to evaluate `last (isort [n,n-1..1])`. Does your answer to the previous question still apply? Explain why or why not.

1. Call-by-value reduction:

```
head (isort (3:2:1:[]))
= head (ins 3 (isort (2:1:[])))
= head (ins 3 (ins 2 (isort (1:[]))))
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1 : ins 2 []))
= head (ins 3 (1 : 2 : []))
= head (1 : ins 3 (2 : []))
= head (1 : 2 : ins 3 [])
= head (1 : 2 : 3 : [])
```

Call-by-name reduction:

```
head (isort (3:2:1:[]))
= head (ins 3 (isort (2:1:[])))
= head (ins 3 (ins 2 (isort (1:[]))))
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1 : ins 2 []))
= head (1 : ins 3 (ins 2 []))
= 1
```

The first 5 steps are the same under both evaluation strategies. However, after that call-by-name evaluation gets to the final result without comparing the numbers 2 and 3. So the number of evaluation steps is lower for call-by-name evaluation. The size of intermediate expressions is the same.

1. Call-by-value evaluation performs `(n-1)+(n-2)+...+1 = n*(n-1)/2` comparisons, while call-by-name evaluation only performs `n-1` comparisons. So the complexity of evaluating the expression is O(n^2) under call-by-value evaluation, but O(n) under call-by-name evaluation.
2. No, in this case both evaluation strategies use the same number of comparisons (`n*(n-1)/2`). The reason is that the `last` function is defined by going through all elements of the list and returning the last, so the whole sorted list has to be computed under either strategy.

Evaluating primes

> Consider the following Haskell functions:
>
> ```
> lookup :: Int -> [a] -> a
> lookup _ []     = error "Index out of range!"
> lookup 0 (x:xs) = x
> lookup n (x:xs) = lookup (n-1) xs
>
> filt :: Integer -> [Integer] -> [Integer]
> filt x (y:ys)
> ```

```
    | y `mod` x == 0  = filt x ys
    | otherwise       = y : filt x ys

  sieve :: [Integer] -> [Integer]
  sieve (x:xs) = x:sieve (filt x xs)

  primes = sieve [2..]
```

Write down the evaluation sequence of `lookup 2 primes` under call-by-name and call-by-value. If the evaluation sequence takes more than 12 steps, you only need to write down the first 12.

To format your answer, please write each evaluation sequence between triple backticks ```, and write only one expression per line. You should not write separate steps for evaluating syntactic sugar for lists, i.e. you may assume that `[2..]` is the same expression as `2:3:4:5:...` without separate steps.

Do you notice a problem when evaluating this expression using call-by-name or call-by-value? What needs to be changed to the definitions of `sieve` and/or `primes` to solve this problem?

Note. When writing down the evaluation sequences, you do not need to write intermediate steps for evaluation of functions from the Haskell prelude (such as `(-)` and `mod`).

Call-by-name:

```
lookup 2 primes
lookup 2 (sieve [2..])
lookup 2 (2:sieve (filt 2 [3..]))
lookup 1 (sieve (filt 2 [3..]))
lookup 1 (sieve (3:filt 2 [4..]))
lookup 1 (3:sieve (filt 3 (filt 2 [4..])))
lookup 0 (sieve (filt 3 (filt 2 [4..])))
lookup 0 (sieve (filt 3 (filt 2 [5..])))
lookup 0 (sieve (filt 3 (5:filt 2 [6..])))
lookup 0 (sieve (5:filt 3 (filt 2 [6..])))
lookup 0 (5:sieve (filt 3 (filt 2 [6..])))
5
```

Call-by-value:

```
lookup 2 primes
lookup 2 (sieve [2..])
lookup 2 (2:sieve (filt 2 [3..]))
lookup 2 (2:sieve (3:(filt 2 [4..])))
lookup 2 (2:sieve (3:(filt 2 [5..])))
lookup 2 (2:sieve (3:5:(filt 2 [6..])))
lookup 2 (2:sieve (3:5:(filt 2 [7..])))
```

```
lookup 2 (2:sieve (3:5:7:(filt 2 [8..])))
lookup 2 (2:sieve (3:5:7:(filt 2 [9..])))
lookup 2 (2:sieve (3:5:7:9:(filt 2 [10..])))
lookup 2 (2:sieve (3:5:7:9:(filt 2 [11..])))
lookup 2 (2:sieve (3:5:7:9:11:(filt 2 [12..])))
...
```

Problem: evaluation under call-by-value loops forever. To fix the problem, we need to change the definition of `primes` to add a maximal element to the list, i.e. `primesUpTo k = sieve [2..k]` .

Week 6A: Agda basica

Half again

> Define the Agda function `halve : Nat → Nat` that computes the result of dividing the given number by 2 (rounded down).

Solution:

```
open import library

halve : Nat → Nat
halve zero          = zero
halve (suc zero)    = zero
halve (suc (suc n)) = suc (halve n)
```

Spec test:

```
open import Agda.Builtin.Equality

test-halve0 : halve 0 ≡ 0
test-halve0 = refl

test-halve1 : halve 1 ≡ 0
test-halve1 = refl

test-halve8 : halve 8 ≡ 4
test-halve8 = refl

test-halve13 : halve 13 ≡ 6
test-halve13 = refl
```

More multiplication

> Define the Agda function `_*_ : Nat → Nat` for multiplication of two natural numbers.

Solution:

```
open import library

_*_ : Nat → Nat → Nat
zero * n = zero
(suc m) * n = (m * n) + n
```

Spec test:

```
open import Agda.Builtin.Equality

test-*0 : 5 * 0 ≡ 0
test-*0 = refl

test-*1 : 5 * 1 ≡ 5
test-*1 = refl

test-*8 : 5 * 8 ≡ 40
test-*8 = refl

test-*13 : 0 * 13 ≡ 0
test-*13 = refl
```

Boolean operators

> Define the type `Bool` with constructors `true` and `false`, and define the functions for negation `not : Bool → Bool`, conjunction `_&&_ : Bool → Bool → Bool`, and disjunction `_||_ : Bool → Bool → Bool` by pattern matching.

Solution:

```
data Bool : Set where
  true  : Bool
  false : Bool

not : Bool → Bool
not true  = false
not false = true

_&&_ : Bool → Bool → Bool
true  && b2 = b2
false && b2 = false
```

```
_||_ : Bool → Bool → Bool
true  || b2 = true
false || b2 = b2
```

Spec test:

```
open import Agda.Builtin.Equality

test-true : Bool
test-true = true

test-false : Bool
test-false = false

test-not-true : not true ≡ false
test-not-true = refl

test-not-false : not false ≡ true
test-not-false = refl

test-and-true-true : true && true ≡ true
test-and-true-true = refl

test-and-true-false : true && false ≡ false
test-and-true-false = refl

test-and-false-true : false && true ≡ false
test-and-false-true = refl

test-and-false-false : false && false ≡ false
test-and-false-false = refl

test-or-true-true : true || true ≡ true
test-or-true-true = refl

test-or-true-false : true || false ≡ true
test-or-true-false = refl

test-or-false-true : false || true ≡ true
test-or-false-true = refl

test-or-false-false : false || false ≡ false
test-or-false-false = refl
```

A list of List functions

> Implement the following Agda functions on lists:

> - `length : {A : Set} → List A → Nat`
> - `_++_ : {A : Set} → List A → List A → List A`
> - `map : {A B : Set} → (A → B) → List A → List B`

Solution:

```
open import library

length : {A : Set} → List A → Nat
length []         = 0
length (x :: xs)  = suc (length xs)

_++_ : {A : Set} → List A → List A → List A
[]          ++  ys  = ys
(x :: xs)  ++  ys  = x :: (xs ++ ys)

map : {A B : Set} → (A → B) → List A → List B
map f []         = []
map f (x :: xs)  = f x :: map f xs
```

Spec test:

```
open import Agda.Builtin.Equality

test-length-nil : {A : Set} → length {A} [] ≡ 0
test-length-nil = refl

test-length-cons : {A : Set} {x : A} {xs : List A} → length (x :: xs) ≡ suc (length
test-length-cons = refl

test-++-nil : {A : Set} {ys : List A} → [] ++ ys ≡ ys
test-++-nil = refl

test-++-cons : {A : Set} {x : A} {xs ys : List A} → (x :: xs) ++ ys ≡ x :: (xs ++ ys
test-++-cons = refl

test-map-nil : {A B : Set} {f : A → B} → map f [] ≡ []
test-map-nil = refl

test-map-cons : {A B : Set} {f : A → B} {x : A} {xs : List A} → map f (x :: xs) ≡ f
test-map-cons = refl
```

Maybe do this exercise?

> Implement the type `Maybe A` with two constructors `just : A → Maybe A` and `nothing : Maybe A`. Next, implement the function `lookup : {A : Set} → List A → Nat → Maybe A`

> that returns `just` the element at the given position in the list if it exists, or `nothing` otherwise.

Solution:

```
open import library

data Maybe (A : Set) : Set where
  just    : A → Maybe A
  nothing : Maybe A

lookup : {A : Set} → List A → Nat → Maybe A
lookup []        _       = nothing
lookup (x :: xs) zero    = just x
lookup (x :: xs) (suc n) = lookup xs n
```

Spec test:

```
open import Agda.Builtin.Equality

test-just : {A : Set} → A → Maybe A
test-just x = just x

test-nothing : {A : Set} → Maybe A
test-nothing = nothing

test-lookup-empty-zero : {A : Set} → lookup {A} [] zero ≡ nothing
test-lookup-empty-zero = refl

test-lookup-empty-suc : {A : Set} {n : Nat} → lookup {A} [] (suc n) ≡ nothing
test-lookup-empty-suc = refl

test-lookup-cons-zero : {A : Set} {x : A} {xs : List A} → lookup (x :: xs) zero ≡ ju
test-lookup-cons-zero = refl

test-lookup-cons-suc : {A : Set} {x : A} {xs : List A} {n : Nat} → lookup (x :: xs)
test-lookup-cons-suc = refl
```

Either left or right

> Define the `Either` type in Agda with constructors `left` and `right`, and implement the higher-order function `cases : {A B C : Set} → Either A B → (A → C) → (B → C) → C`.

Solution:

```
data Either (A B : Set) : Set where
```

```
    left   : A → Either A B
    right  : B → Either A B

  cases : {A B C : Set} → Either A B → (A → C) → (B → C) → C
  cases (left x)   f  g  = f x
  cases (right y)  f  g  = g y
```

Spec test:

```
open import Agda.Builtin.Equality

test-left-type : {A B : Set} → A → Either A B
test-left-type = left

test-right-type : {A B : Set} → B → Either A B
test-right-type = right

test-cases-type : {A B C : Set} → Either A B → (A → C) → (B → C) → C
test-cases-type = cases

test-cases-left : {A B C : Set} {x : A} {f : A → C} {g : B → C} → cases (left x) f g
test-cases-left = refl

test-cases-right : {A B C : Set} {y : B} {f : A → C} {g : B → C} → cases (right y) f
test-cases-right = refl
```

Week 6B: Dependent Types

Going down fast

> Implement the function `downFrom : (n : Nat) → Vec Nat n` that produces the vector `(n-1) :: (n-2) :: ... :: 0`.

Solution:

```
open import library

downFrom : (n : Nat) → Vec Nat n
downFrom zero    = []
downFrom (suc n) = n :: downFrom n
```

Spec test:

```
open import Agda.Builtin.Equality
```

```
test-downFrom-type : (n : Nat) → Vec Nat n
test-downFrom-type = downFrom

test-downFrom-three : {A : Set} → downFrom 3 ≡ (2 :: 1 :: 0 :: [])
test-downFrom-three = refl

test-downFrom-zero : {A : Set} → downFrom zero ≡ []
test-downFrom-zero = refl

test-downFrom-suc : {A : Set} {n : Nat} → downFrom (suc n) ≡ n :: downFrom n
test-downFrom-suc = refl
```

Tail risks

> Implement the function `tail : {A : Set} {n : Nat} → Vec A (suc n) → Vec A n`.

Solution:

```
open import library

tail : {A : Set}{n : Nat} → Vec A (suc n) → Vec A n
tail (x :: xs) = xs
```

Spec test:

```
open import Agda.Builtin.Equality

test-tail-type : {A : Set}{n : Nat} → Vec A (suc n) → Vec A n
test-tail-type = tail

test-tail-singleton : {A : Set}{x : A} → tail (x :: []) ≡ []
test-tail-singleton = refl

test-tail-cons : {A : Set}{n : Nat}{x : A}{xs : Vec A n} → tail (x :: xs) ≡ xs
test-tail-cons = refl
```

Putting the dots on the vector

> Implement the function `dotProduct : {n : Nat} → Vec Nat n → Vec Nat n → Nat` that calculates the “dot product” (or scalar product) of two vectors. For example, `dotProduct (a :: b :: c :: []) (x :: y :: z :: []) = a * x + b * y + c * z`. Note that the type of the function enforces the two vectors to have the same length, so you don’t need to write the clauses where that is not the case.

Solution:

```
open import library

dotProduct : {n : Nat} → Vec Nat n → Vec Nat n → Nat
dotProduct []        []        = 0
dotProduct (x :: xs) (y :: ys) = x * y + dotProduct xs ys
```

Spec test:

```
open import Agda.Builtin.Equality

test-dotProduct-type : {n : Nat} → Vec Nat n → Vec Nat n -> Nat
test-dotProduct-type = dotProduct

test-dotProduct-single : {A : Set}{x y : Nat} → dotProduct (x :: []) (y :: []) ≡ x *
test-dotProduct-single = refl

test-dotProduct-empty : {A : Set}{x : A} → dotProduct [] [] ≡ 0
test-dotProduct-empty = refl

test-dotProduct-cons : {n : Nat}{x y : Nat}{xs ys : Vec Nat n} → dotProduct (x :: xs
test-dotProduct-cons = refl
```

Vector update

> Implement the function `putVec : {A : Set}{n : Nat} → Fin n → A → Vec A n → Vec A n` that sets the value at the given position in the vector to the given value, and leaves the rest of the vector unchanged.

Solution:

```
open import library

putVec : {A : Set}{n : Nat} → Fin n → A → Vec A n → Vec A n
putVec zero    x (y :: ys) = x :: ys
putVec (suc i) x (y :: ys) = y :: putVec i x ys
```

Spec test:

```
open import Agda.Builtin.Equality

test-putVec-type : {A : Set}{n : Nat} → Fin n -> A -> Vec A n → Vec A n
test-putVec-type = putVec
```

```
test-putVec-single : {A : Set}{x y : A} → putVec zero x (y :: []) ≡ (x :: [])
test-putVec-single = refl

test-putVec-here : {A : Set}{n : Nat}{x y : A}{ys : Vec A n} → putVec zero x (y :: y
test-putVec-here = refl

test-putVec-there : {A : Set}{n : Nat}{i : Fin n}{x y : A}{ys : Vec A n} → putVec (s
test-putVec-there = refl
```

Seeing double

> In the Library code, there are two possible implementations of the (non-dependent) pair type in Agda: one direct one as a datatype, and one type alias for the *dependent* pair type where the type of the second component ignores its input. Implement two functions `from : {A B : Set} → A × B → A ×' B` and `to : {A B : Set} → A ×' B → A × B` converting between the two representations.

Solution:

```
open import library

from : {A B : Set} → A × B → A ×' B
from (x , y) = x , y

to : {A B : Set} → A ×' B → A × B
to (x , y) = x , y
```

Spec test:

```
open import Agda.Builtin.Equality

test-from-type : {A B : Set} → A × B → A ×' B
test-from-type = from

test-to-type : {A B : Set} → A ×' B → A × B
test-to-type = to

test-from : {A B : Set} {x : A} {y : B} → from (x , y) ≡ (x , y)
test-from = refl

test-to : {A B : Set} {x : A} {y : B} → from (x , y) ≡ (x , y)
test-to = refl
```

There's lists and there's lists

> In the Library code, there are two possible implementations of the regular list type in Agda: one direct definition as a datatype, and one type alias for a dependent pair of a natural number `n` and a vector of length `n`. Implement two functions `from : {A : Set} → List A → List' A` and `to : {A : Set} → List' A → List A` converting between the two representations.
>
> **Hint.** For the function `from`, first implement functions `[]' : {A : Set} → List' A` and `_::'_ : {A : Set} → A → List' A → List' A`.

Solution:

```
open import library

[]' : {A : Set} → List' A
[]' = 0 , []

_::'_ : {A : Set} → A → List' A → List' A
x ::' (n , xs) = suc n , x :: xs

from : {A : Set} → List A → List' A
from [] = []'
from (x :: xs) = x ::' from xs

to : {A : Set} → List' A → List A
to (zero  , []        ) = []
to (suc n , (x :: xs)) = x :: to (n , xs)
```

Spec test:

```
open import Agda.Builtin.Equality

test-from-type : {A : Set} → List A → List' A
test-from-type = from

test-to-type : {A : Set} → List' A → List A
test-to-type = to

test-from-nil : {A : Set} → from {A} [] ≡ (0 , [])
test-from-nil = refl

test-from-single : {A : Set} {x : A} → from (x :: []) ≡ (1 , (x :: []))
test-from-single = refl

test-from-double : {A : Set} {x1 x2 : A} → from (x1 :: x2 :: []) ≡ (2 , x1 :: x2 ::
test-from-double = refl

test-from-triple : {A : Set} {x1 x2 x3 : A} → from (x1 :: x2 :: x3 :: []) ≡ (3 , x1
test-from-triple = refl
```

```
test-to-nil : {A : Set} → to {A} (0 , []) ≡ []
test-to-nil = refl

test-to-single : {A : Set} {x : A} → to (1 , (x :: [])) ≡ (x :: [])
test-to-single = refl

test-to-cons : {A : Set} {x : A} {n : Nat} {xs : Vec A n} → to (suc n , (x :: xs)) ≡
test-to-cons = refl
```

Week 7A: Curry-Howard Correspondence

Through the lens of Curry-Howard (1)

> Translate the following proposition to Agda types using the Curry-Howard correspondence, and prove the statement by implementing a function of that type:
>
> "If A then (B implies A)"

Solution:

```
open import library

prop1 : {A B : Set} → A → (B → A)
prop1 = λ x y → x
```

Spec test:

```
open import Agda.Builtin.Equality

test-prop1-type : {A B : Set} → A → (B → A)
test-prop1-type = prop1

test-prop1 : {A B : Set} {x : A} {y : B} → prop1 x y ≡ x
test-prop1 = refl
```

Through the lens of Curry-Howard (2)

> Translate the following proposition to Agda types using the Curry-Howard correspondence, and prove the statement by implementing a function of that type:
>
> "If (A and *true*) then (A or *false*)"

Solution:

```
open import library

prop2 : {A : Set} → (A × ⊤) → Either A ⊥
prop2 = λ x → left (fst x)
```

Spec test:

```
open import Agda.Builtin.Equality

test-prop2-type : {A : Set} → (A × ⊤) → Either A ⊥
test-prop2-type = prop2

test-prop2 : {A : Set} {x : A} → prop2 (x , tt) ≡ left x
test-prop2 = refl
```

Through the lens of Curry-Howard (3)

> Translate the following proposition to Agda types using the Curry-Howard correspondence, and prove the statement by implementing a function of that type:
>
> “If A implies (B implies C), then (A and B) implies C”

Solution:

```
open import library

prop3 : {A B C : Set} → (A → (B → C)) → (A × B) → C
prop3 = λ f xy → f (fst xy) (snd xy)
```

Spec test:

```
open import Agda.Builtin.Equality

test-prop3-type : {A B C : Set} → (A → (B → C)) → (A × B) → C
test-prop3-type = prop3

test-prop3 : {A B : Set} {x : A} {y : B} → prop3 _,_ (x , y) ≡ (x , y)
test-prop3 = refl
```

Through the lens of Curry-Howard (4)

> Translate the following proposition to Agda types using the Curry-Howard correspondence, and prove the statement by implementing a function of that type:
>
> “If A and (B or C), then either (A and B) or (A and C)”

Solution:

```
open import library

prop4 : {A B C : Set} → A × (Either B C) → Either (A × B) (A × C)
prop4 = λ x → cases (snd x) (λ y → left (fst x , y)) λ z → right (fst x , z)
```

Spec test:

```
open import Agda.Builtin.Equality

test-prop4-type : {A B C : Set} → A × (Either B C) → Either (A × B) (A × C)
test-prop4-type = prop4

test-prop4-left : {A B C : Set} {x : A} {y : B} → prop4 (x , left {B} {C} y) ≡ left
test-prop4-left = refl

test-prop4-right : {A B C : Set} {x : A} {z : C} → prop4 (x , right {B} {C} z) ≡ rig
test-prop4-right = refl
```

Through the lens of Curry-Howard (5)

> Translate the following proposition to Agda types using the Curry-Howard correspondence, and prove the statement by implementing a function of that type:
>
> “If A implies C and B implies D, then (A and B) implies (C and D)”

Solution:

```
open import library

prop5 : {A B C D : Set} → (A → C) × (B → D) → A × B → C × D
prop5 = λ fg xy → fst fg (fst xy) , snd fg (snd xy)
```

Spec test:

```
open import Agda.Builtin.Equality

test-prop5-type : {A B C D : Set} → (A → C) × (B → D) → A × B → C × D
test-prop5-type = prop5
```

```
test-prop5 : {A B C D : Set} {f : A → C} {g : B → D} {x : A} {y : B}
           → prop5 (f , g) (x , y) ≡ (f x , g y)
test-prop5 = refl
```

Bonus: That's not not true!

> Since Agda uses a constructive logic, it is not possible to prove non-constructive statements such as “for all P, either P or not P” (also known as the *law of the excluded middle*). However, we can prove the double negation of this statement: it is *not not* the case that for all P, either P or not P. To show that this double negation translation indeed works, prove this statement in Agda by implementing a function of type `(Either P (P → ⊥) → ⊥) → ⊥`.

Solution:

```
open import library

f : {P : Set} → (Either P (P → ⊥) → ⊥) → ⊥
f h = h (right (λ x → h (left x)))
```

Spec test:

```
open import Agda.Builtin.Equality

test-f-type : {P : Set} → (Either P (P → ⊥) → ⊥) → ⊥
test-f-type h = f h
```

Week 7B: Equational reasoning in Agda

Replication replication replication...

> Consider the following function:
>
> ```
> replicate : {A : Set} → Nat → A → List A
> replicate zero    x = []
> replicate (suc n) x = x :: replicate n x
> ```
>
> Complete the proof that the length of `replicate n x` is always equal to `n`, by filling in the holes `?`.

Solution:

```
open import library

length-replicate : {A : Set} → (n : Nat) (x : A) → length (replicate n x) ≡ n
length-replicate {A} zero x =
  begin
    length (replicate zero x)
  =⟨⟩                            -- applying replicate
    length {A} []
  =⟨⟩                            -- applying length
    zero
  end
length-replicate (suc n) x =
  begin
    length (replicate (suc n) x)
  =⟨⟩                                -- applying replicate
    length (x :: replicate n x)
  =⟨⟩                                -- applying length
    suc (length (replicate n x))
  =⟨ cong suc (length-replicate n x) ⟩  -- using induction hypothesis
    suc n
  end
```

Spec test:

```
open import library

test-length-replicate-type : {A : Set} → (n : Nat) (x : A) → length (replicate n x)
test-length-replicate-type = length-replicate
```

Reasoning about addition

> In the lecture notes, it is proven that `n + zero` equals `n` for all natural numbers `n`. Following the example of this proof, now write down a proof that `m + (suc n)` is equal to `suc (m + n)` for all natural numbers `m` and `n`.
>
> **Bonus question.** Now write down a proof of commutativity of addition: `m + n` equals `n + m` for all natural numbers `m` and `n`, by making use of the previous two lemmas.

Solution:

```
open import library

add-n-zero : (n : Nat) → n + zero ≡ n
add-n-zero zero =
  begin
    zero + zero
```

```
    =⟨⟩                             -- applying +
      zero
    end
  add-n-zero (suc n) =
    begin
      (suc n) + zero
    =⟨⟩                             -- applying +
      suc (n + zero)
    =⟨ cong suc (add-n-zero n) ⟩    -- using induction hypothesis
      suc n
    end

  add-n-suc : (m n : Nat) → m + (suc n) ≡ suc (m + n)
  add-n-suc zero    n =
    begin
      zero + (suc n)
    =⟨⟩                            -- applying +
      suc n
    end
  add-n-suc (suc m) n =
    begin
      (suc m) + (suc n)
    =⟨⟩                            -- applying +
      suc (m + (suc n))
    =⟨ cong suc (add-n-suc m n) ⟩  -- using induction hypothesis
      suc (suc (m + n))
    end

  -- Bonus part: prove commutativity of addition.
  add-comm : (m n : Nat) → m + n ≡ n + m
  add-comm zero    n =
    begin
      zero + n
    =⟨⟩
      n
    =⟨ sym (add-n-zero n) ⟩
      n + zero
    end
  add-comm (suc m) n =
    begin
      (suc m) + n
    =⟨⟩
      suc (m + n)
    =⟨ cong suc (add-comm m n) ⟩
      suc (n + m)
    =⟨ sym (add-n-suc n m) ⟩
      n + (suc m)
    end
```

Spec test:

```
open import library

test-add-n-suc-type : (m n : Nat) → m + (suc n) ≡ suc (m + n)
test-add-n-suc-type = add-n-suc

test-add-comm-type : (m n : Nat) → m + n ≡ n + m
test-add-comm-type = add-comm
```

Length of map

> Prove that using `map` does not change the length of a list, i.e. that `length (map f xs)` is equal to `length xs`.

Solution:

```
open import library

length-map : {A B : Set} (f : A → B) (xs : List A) → length (map f xs) ≡ length xs
length-map {A} {B} f [] =
  begin
    length (map f [])
  =⟨⟩
    length {B} []
  end
length-map f (x :: xs) =
  begin
    length (map f (x :: xs))
  =⟨⟩
    length (f x :: map f xs)
  =⟨⟩
    suc (length (map f xs))
  =⟨ cong suc (length-map f xs) ⟩
    suc (length xs)
  =⟨⟩
    length (x :: xs)
  end
```

Spec test:

```
open import library

test-length-map-type : {A B : Set} (f : A → B) (xs : List A) → length (map f xs) ≡ l
test-length-map-type = length-map
```

Append nothing

> Prove that `xs ++ []` is equal to `xs` (see Library code for the definition of `_++_`).

Solution:

```
open import library

append-[] : {A : Set} → (xs : List A) → xs ++ [] ≡ xs
append-[] [] =
  begin
    [] ++ []
  =⟨⟩
    []
  end
append-[] (x :: xs) =
  begin
    (x :: xs) ++ []
  =⟨⟩
    x :: (xs ++ [])
  =⟨ cong (x ::_) (append-[] xs) ⟩
    x :: xs
  end
```

Spec test:

```
open import library

test-append-[]-type : {A : Set} → (xs : List A) → xs ++ [] ≡ xs
test-append-[]-type = append-[]
```

Append more

> Prove that `(xs ++ ys) ++ zs` is equal to `xs ++ (ys ++ zs)` (see Library code for the definition of `_++_`).

Solution:

```
open import library

append-assoc : {A : Set} → (xs ys zs : List A)
             → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
append-assoc [] ys zs =
  begin
    ([] ++ ys) ++ zs
  =⟨⟩                                         -- applying inner ++
    ys ++ zs
  =⟨⟩                                         -- unapplying ++
```

```
    [] ++ (ys ++ zs)
  end
append-assoc (x :: xs) ys zs =
  begin
    ((x :: xs) ++ ys) ++ zs
  =⟨⟩                                         -- applying inner ++
    (x :: (xs ++ ys)) ++ zs
  =⟨⟩                                         -- applying outer ++
    x :: ((xs ++ ys) ++ zs)
  =⟨ cong (x ::_) (append-assoc xs ys zs) ⟩   -- using induction hypothesis
    x :: (xs ++ (ys ++ zs))
  =⟨⟩                                         -- unapplying outer ++
    (x :: xs) ++ (ys ++ zs)
  end
```

Spec test:

```
open import library

test-append-assoc-type : {A : Set} → (xs ys zs : List A)
                       → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
test-append-assoc-type = append-assoc
```

Take it or leave it

> Define the functions `take` and `drop` that respectively return or remove the first `n` elements of the list (or all elements if the list is shorter).
>
> Next, prove that for any number `n` and any list `xs`, we have `take n xs ++ drop n xs = xs`.

Solution:

```
open import library

take : {A : Set} → Nat → List A → List A
take zero    xs        = []
take _       []        = []
take (suc n) (x :: xs) = x :: take n xs

drop : {A : Set} → Nat → List A → List A
drop zero    xs        = xs
drop _       []        = []
drop (suc n) (x :: xs) = drop n xs

take-drop : {A : Set} (n : Nat) (xs : List A) → take n xs ++ drop n xs ≡ xs
take-drop zero    xs        =
```

```
    begin
      take zero xs ++ drop zero xs
    =⟨⟩
      [] ++ drop zero xs
    =⟨⟩
      drop zero xs
    =⟨⟩
      xs
    end
  take-drop (suc n) []        =
    begin
      take (suc n) [] ++ drop (suc n) []
    =⟨⟩
      [] ++ drop (suc n) []
    =⟨⟩
      drop (suc n) []
    =⟨⟩
      []
    end
  take-drop (suc n) (x :: xs) =
    begin
      take (suc n) (x :: xs) ++ drop (suc n) (x :: xs)
    =⟨⟩
      (x :: take n xs) ++ drop (suc n) (x :: xs)
    =⟨⟩
      (x :: take n xs) ++ drop n xs
    =⟨⟩
      x :: (take n xs ++ drop n xs)
    =⟨ cong (x ::_) (take-drop n xs) ⟩
      x :: xs
    end
```

Spec test:

```
  open import library

  test-take-type : {A : Set} → Nat → List A → List A
  test-take-type = take

  test-take-none : {A : Set} {x : A} {xs : List A} → take 0 (x :: xs) ≡ []
  test-take-none = refl

  test-take-one : {A : Set} {x1 x2 : A} {xs : List A} → take 1 (x1 :: x2 :: xs) ≡ x1 :
  test-take-one = refl

  test-drop-type : {A : Set} → Nat → List A → List A
  test-drop-type = drop

  test-drop-none : {A : Set} {x : A} {xs : List A} → drop 0 (x :: xs) ≡ (x :: xs)
  test-drop-none = refl
```

```
test-drop-one : {A : Set} {x1 x2 : A} {xs : List A} → drop 1 (x1 :: x2 :: xs) ≡ x2 :
test-drop-one = refl

test-take-drop-type : {A : Set} (n : Nat) (xs : List A) → take n xs ++ drop n xs ≡ x
test-take-drop-type = take-drop
```

Two ways to flatten

> In the lecture notes, there are two different definitions of the function `flatten` on `Tree` s: a direct one, and one using an accumulator. Prove that the two definitions are equivalent, i.e. that `flatten' t = flatten t` for every `t : Tree A` . You can use the given proof of `flatten-acc-flatten` as well as the `append-assoc` lemma from the library code.

Solution:

```
open import library

flatten'-flatten : {A : Set} → (t : Tree A) → flatten' t ≡ flatten t
flatten'-flatten t =
  begin
    flatten' t
  =⟨⟩
    flatten-acc t []
  =⟨ flatten-acc-flatten t [] ⟩
    flatten t ++ []
  =⟨ append-[] (flatten t) ⟩
    flatten t
  end
```

Spec test:

```
open import library

test-flatten'-flatten-type : {A : Set} → (t : Tree A) → flatten' t ≡ flatten t
test-flatten'-flatten-type = flatten'-flatten
```