Re-exam – Introduction to Functional Programming

TDA555/DIT441 (DIT440), HT-22
Chalmers and Göteborgs Universitet, CSE

Day: 2023-01-03, *Time:* 14:00-18:00, *Place:* Johanneberg

Course responsible
Prof. Dave Sands (0737207663). He will visit the exam room once between 15:00 and 16:00, and is after that available by phone.

Allowed aids
An English dictionary.

Grading
The exam consist of two parts: a part with seven small assignments and a part with two more advanced assignments; there are in total nine assignments.

- To pass the exam (with a 3) you need to give good enough answers for five out of the nine assignments. An answer with minor mistakes might be accepted, but this is at the discretion of the marker. An answer with large mistakes will be marked as incorrect.
- You do not need to solve the assignments from part II to pass the exam and you are happy with a 3! You are though encouraged to try the assignments from part II: they count to pass the exam, and you may get a higher grade.
- For a 4 you need to pass Part I (five out of seven assignments) and one assignment of your choice from Part II.
- For a 5 you need to pass Part I (five out of seven assignments) and both assignments from Part II.

Notes

– Begin each assignment on a new sheet and write your number on it.

– You may write your answers in Swedish and English.

– Excessively complicated answers might be rejected.

– *Write legibly!* Solutions that are difficult to read are marked as incorrect!

– You can make use of the standard Haskell functions and types given in the attached list (you have to implement other functions yourself if you want to use them). You do not have to import standard modules in your solutions. You do not have to copy any of the code provided.

– **Good luck!**

Part I

1

Given the following definitions:

```
f n xs = map (g n) xs

g n x | n > 0      = x : g (n - 1) x
      | otherwise = []
```

a) What does the expression `f 2 "A"` evaluate to? Write down the intermediate steps of your computation. If the type of your answer is incorrect then your solution will be considered incorrect.

b) What are the most general types (type signatures) of `f` and `g`?

2

Pell numbers are an infinite sequence of integers and are similar to Fibonacci numbers. The n-th Pell number is defined by the following relation:

$$P_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ 2P_{n-1} + P_{n-2} & \text{otherwise.} \end{cases}$$

That is, the sequence of Pell numbers starts with 0 and 1, and then each Pell number is the sum of twice the previous Pell number and the Pell number before that.

Your tasks are:

a) First, write a function that calculates the n-th Pell number:

```
pell :: Int -> Integer
```

b) Next, we can use Pell numbers to *approximate* the square root of two ($\sqrt{2}$). The following formula can be used for this approximation:

$$\frac{P_{n-1} + P_n}{P_n}$$

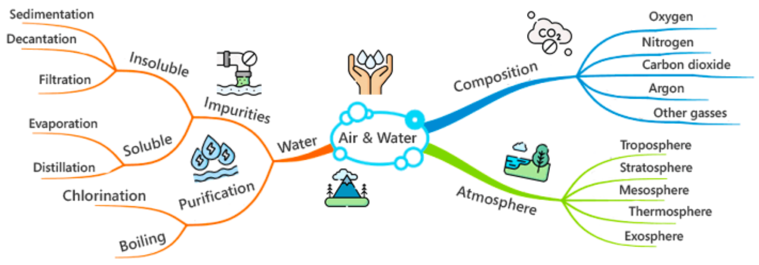
where the numerator is the sum of the n-th Pell number (P_n) and its predecessor (P_{n-1}), and the denominator is the n-th Pell number. Define a function that calculates the approximation of the square root of two using the above formula:

```
approx :: Int -> Double
```

The function takes the n-th number as input and returns the approximation as a value of type `Double`. *Hint:* the function `fromIntegral` can be used to convert an integer to a `Double`.

3

A *mind map*[1] can be used to visualize information or ideas about a concept in a hierarchal manner. For example[2]:



We can model a mind map in Haskell with the following recursive datatype:

```
data MindMap = Topic String | Branch String [MindMap]
```

where a mind map is either a single topic, or it is a topic that is associated with a number of other topics, which are stored in a list. Note that a branch also stores information (as a string). The example 'Air & Water' mind map that is depicted above can be expressed (partly) as follows:

```
airWater :: MindMap
airWater = Branch "Air & Water" [water, compo, atmos]
 where
   water = Branch "Water"       [Branch "Impurities" [], Branch "Puri..." []]
   compo = Branch "Composition" [Topic "Oxygen", Topic "Nitrogen"]
   atmos = Branch "Atmosphere"  [Topic "Troposphere", Topic "Strato..."]
```

Your task is to implement the following higher-order function:

```
mapMindMap :: (String -> String) -> MindMap -> MindMap
```

that applies a given function to every string in the mind map, that is, both to a string in a `Topic` as well as in a `Branch`. For example, the next GHCi interaction shows the result of applying `(map toUpper)` to the example 'Air & Water' mind map:

```
ghci> mapMindMap (map toUpper) airWater
Branch "AIR & WATER"
  [Branch "WATER" [Branch "IMPURITIES" [],Branch "PURI..." []],
   Branch "COMPOSITION" [Topic "OXYGEN",Topic "NITROGEN"],
   Branch "ATMOSPHERE" [Topic "TROPOSPHERE",Topic "STRATO..."]]
```

[1]Svenska: tankekarta
[2]Source: http://mindmapping.com

4

Consider the following datatype in Haskell that models a person:

```
data Person = Person
  { name    :: String
  , age     :: Int
  , partner :: Maybe String
  } deriving Show
```

Your task is to define an `IO` function that asks a user for a name, age, if there exists a partner, and if so, asks for another name. The function should read all these values, construct a value type `Person` and return this. The function has the following type signature:

```
readPerson :: IO Person
```

The following excerpt shows an example interaction:

```
ghci> readPerson
Name: Alex
Age: 44
Partner (y/n): y
Partner name: Anita
Person {name = "Alex", age = 44, partner = Just "Anita"}
```

Note that the function *returns* a person, it does not print it. It is GHCi that displays the result in the above excerpt.

5

Consider the following data type definition that models an electronic billboard:

```
type Pixel = (Int, Int)

data BillBoard = BB { size :: (Int, Int), actives :: [Pixel] }
```

A billboard, which can be regarded as a matrix of pixels, consists of its size and a list of active pixels. The size field of a billboard is a tuple of integers where the first element is the number of rows, and the second element the number of columns. A pixel is a pair of integers which denotes its place (row and column) on the billboard. For example, `(0, 3)` is found on the first row and fourth column. We use zero-based indexing, that is, index `(0, 0)` points to the pixel on the first row and first column.

Using the above data definition we can make an example billboard:

```
lambda :: BillBoard
lambda = BB (4, 10) [(0,2),(1,3),(2,2),(2,4),(3,1),(3,5)]
```

which can be textually represented as follows:

```
..#.......
...#......
..#.#.....
.#...#....
```

where an active pixel is represented by the character `'#'`, and a non-active pixel by a dot `'.'`.

Your tasks are:

a) Write a function that validates that all active pixels are in range with respect to the size of the billboard.

```
valid :: BillBoard -> Bool
```

b) Define a function that *moves* all the active pixels in a billboard a given number of steps (columns) to the right:

```
move :: Int -> BillBoard -> BillBoard
```

You need to make sure that all active pixels are valid, that is, are in range with respect to the billboard's size. In other words, when moving an active pixel on the last (most right) column a step to the right, it is removed from the billboard.

The following example shows the textual representation of calling `move 5` on `lambda`:

```
.......#..
........#.
.......#.#
......#...
```

6

In assignment 2 we introduced *Pell numbers* and in this assignment you are going to define some QuickCheck properties on Pell numbers. Note that you can complete this assignment without implementing assignment 2, just assume there exists a function:

```
pell :: Int -> Integer
```

that works correctly and returns the n-th Pell number.

Your tasks in this assignment are:

a) Write a property that validates that the n-th Pell number is *larger* than its predecessor (the $(n-1)$-th Pell number):

```
prop_larger :: Int -> Bool
```

Mind you that Pell numbers grow exponentially and get very large, very quickly. You need to restrain the input value (n) to be within the range 0 to 50. In all other cases, that is, if the input is either negative or larger than 50, the property returns `True`.

b) The following formula holds for Pell numbers:

$$P_{n+1}P_{n-1} - {P_n}^2 = (-1)^n$$

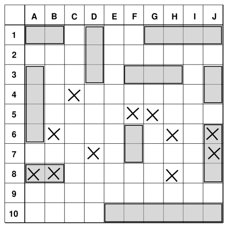
Implement a property that checks this:

```
prop_identity :: Int -> Bool
```

Make sure again that the Pell numbers don't get too large.

7

Your taks is to model the game 'Battleship' (Sänka skepp) in Haskell. The game has two players, which have a name and total number of points, and consists of two boards on which each of the player's ships are placed. During the run of the game the players shoot at a particular position on the other player's board, hoping to hit a ship.



A board is a ten by ten grid and each cell needs to store whether a part of a ship is placed on it, and whether or not it has been shot. The figure[3] to the right illustrates this.

A ship can be one of the following: Carrier, Battleship, Destroyer, Submarine, or Patrol Boat.

[3]Source: Wikipedia

Part II

8

In this assignment you will implement (a part of) a *line editor*, which reads keyboard presses and produces a corresponding text string. Keyboard presses are represented by the following data type:

```haskell
data Key = Chr Char | Del | GoLeft | GoRight | Copy Int | Paste
  deriving (Eq, Show)
```

The semantics of the keyboard presses are:

- `Chr`: represents a normal visible character,
- `Del`: represents the deletion key, which deletes the character to the left of the cursor (if possible),
- `GoLeft`: moves the cursor to the left (if possible),
- `GoRight`: moves the cursor to the right (if possible),
- `Copy`: copies the a given number of characters to the *left* of the cursor (as many as are present) and stores them in a clipboard,
- `Paste`: pastes the characters stored in the clipboard (if any) to the *right* of the cursor (the cursor remains at the same position).

Your task is to write the following function:

```haskell
run :: [Key] -> String
```

that given a list of keys computes the resulting text string. For example:

```
ghci> run [Chr 'f', Chr 'p', Chr 'x', Del, Copy 2, GoLeft, Paste]
"ffpp"
```

Hint: avoid using the indexing operator `(!!)`, it is a good idea to represent the current line as two separate lists: the part to the left and the part to the right of the cursor.

9

The *HyperText Markup Language* (HTML) is a language for describing documents. All webpages are written using HTML. Documents written in HTML have a structure that is determined by the use of *tags*. We can enclose a particular part of our document within certain tags, to indicate this structure. To enclose a part of a document in tags, we use matching open tags and close tags. For example:

- Text enclosed in boldface tags `<B> ... </B>` indicates that the text should be in boldface. Here, `<B>` is the open tag, and `</B>` is the corresponding close tag.
- Text enclosed in emphasize tags `<EM> ... </EM>` indicates that the text should be emphasized (often using italics).
- Text enclosed in paragraph tags `<P> ... </P>` indicates that the text forms a paragraph (often by having an empty line before and after).

(In reality, tags contain more information than just the tag name (such as `B`, `EM`, `P`, etc.), but for simplicity we do not deal with that here.) Here is an example of HTML code:

```
Welcome to my website!
<P>
  <B>
    My hobbies are <EM>Haskell</EM> programming and playing <EM>Myst</EM>.
  </B>
</P>
<P>
  Thanks for visiting! <EM>anna@gmail.com</EM>.
  <P>
    Bye, bye!
  </P>
</P>
```

and here is what it would look like in a browser:

Welcome to my website!

My hobbies are *Haskell* programming and playing *Myst*.

Thanks for visiting! *anna@gmail.com*

Bye, bye!

We can represent HTML documents in Haskell as a list of tags:

```
type HTML = [Tag]
```

There are three different kinds of tags: a piece of text, an open tag, and a close tag.

```
data Tag = Text String | Open  String | Close String deriving (Eq, Show)
```

The example piece of HTML above can be represented by the following Haskell expression:

```haskell
annasSida :: HTML
annasSida =
  [ Text "Welcome to my website!"
  , Open "P"
    , Open "B"
      , Text "My hobbies are ", Open "EM", Text "Haskell", Close "EM"
      , Text " programming and playing ", Open "EM", Text "Myst", Close "EM"
      , Text "."
    , Close "B"
  , Close "P"
  , Open "P"
    , Text "Thanks for visiting! ", Open "EM", Text "anna@gmail.com", Close "EM"
    , Open "P", Text ". Bye, bye!", Close "P"
  , Close "P"
  ]
```

Your task is to define the following function:

```haskell
within :: String -> HTML -> [[Tag]]
```

that returns the parts of the HTML document that are enclosed within a given tag. The parts should include the open and close tag. Note that tags can be nested, which means that some parts may appear multiple times in the output list. For example, calling `within` to retrieve the parts enclosed within `EM` tags, results in a list with three HTML parts:

```
ghci> within "EM" annasSida
[[Open "EM",Text "Haskell",Close "EM"],[Open "EM",Text "Myst",Close "EM"],
[Open "EM",Text "anna@gmail.com",Close "EM"]]
```

Calling the `within` function with the tag `P` on the same HTML document gives the following answer:

```
ghci> within "P" annasSida
[[Open "P",Open "B",Text "My hobbies are ",Open "EM",Text "Haskell",Close "EM",
Text " programming and playing ",Open "EM",Text "Myst",Close "EM",Text ".",
Close "B",Close "P"],[Open "P",Text ". Bye, bye!",Close "P"],[Open "P",
Text "Thanks for visiting! ",Open "EM",Text "anna@gmail.com",Close "EM",
Text ". Bye, bye!",Close "P"]]
```

where you can see that the text ". Bye, bye!" appears twice in the output list, the first time because it is enclosed within `P` tags, and a second time because it is enclosed in `P` tags on a higher level.

You may assume that the input HTML is well-formed and don't need to validate the input nor take care of invalid input.

```haskell
{- This is a list of selected functions from the
   standard Haskell modules: Prelude Data.List
   Data.Maybe Data.Char Control.Monad         -}
---------------------------------------------
-- * Standard type classes

class Show a where  show :: a -> String

class Read a where  read :: String -> a

class Eq a where
  (==), (/=) :: a -> a -> Bool

class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class (Eq a, Show a) => Num a where
  (+), (-), (*)    :: a -> a -> a
  negate           :: a -> a
  abs, signum      :: a -> a
  fromInteger      :: Integer -> a

class (Num a, Ord a) => Real a where
  toRational       ::  a -> Rational

class (Real a, Enum a) => Integral a where
  quot, rem        :: a -> a -> a
  div, mod         :: a -> a -> a
  toInteger        :: a -> Integer

class Num a => Fractional a where
  (/)              :: a -> a -> a
  fromRational     :: Rational -> a

class (Fractional a) => Floating a where
  exp, log, sqrt      :: a -> a
  sin, cos, tan       :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round :: (Integral b) => a -> b
  ceiling, floor  :: (Integral b) => a -> b

---------------------------------------------
-- * Numerical functions

even, odd        :: Integral a => a -> Bool
even n           = n 'rem' 2 == 0
odd              = not . even

---------------------------------------------
-- * Monadic functions
sequence     :: Monad m => [m a] -> m [a]
sequence     = foldr mcons (return [])
  where mcons p q = do x <- p
                       xs <- q
                       return (x:xs)

sequence_    :: Monad m => [m a] -> m ()
sequence_ xs = do sequence xs
                  return ()

liftM   :: Monad m => (a -> b) -> m a -> m b
liftM f m1   = do x1 <- m1
                  return (f x1)
---------------------------------------------
```

```haskell
-- * Functions on functions
id               :: a -> a
id x             = x

const            :: a -> b -> a
const x _        = x

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

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

($)              :: (a -> b) -> a -> b
f $ x            = f x
-----------------------------------------
-- * Functions on Bools
data Bool = False | True

(&&), (||)        :: Bool -> Bool -> Bool
True  && x        = x
False && _        = False
True  || _        = True
False || x        = x

not               :: Bool -> Bool
not True          = False
not False         = True
-----------------------------------------
-- * Functions on Maybe
data Maybe a = Nothing | Just a

isJust,isNothing       :: Maybe a -> Bool
isJust (Just a)        =  True
isJust Nothing         =  False

isNothing              =  not . isJust

fromJust               :: Maybe a -> a
fromJust (Just a)      =  a

maybeToList            :: Maybe a -> [a]
maybeToList Nothing    =  []
maybeToList (Just a)   =  [a]

listToMaybe            :: [a] -> Maybe a
listToMaybe []         =  Nothing
listToMaybe (a:_)      =  Just a

catMaybes              :: [Maybe a] -> [a]
catMaybes ls           = [x | Just x <- ls]
-----------------------------------------
-- * Functions on pairs
fst               :: (a,b) -> a
fst (x,y)         =  x
snd               :: (a,b) -> b
snd (x,y)         =  y

swap              :: (a,b) -> (b,a)
swap (a,b)        = (b,a)

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

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p    =  f (fst p) (snd p)
```

```haskell
-- * Functions on lists

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

(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) ys xs

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

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

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

head, last        :: [a] -> a
head (x:_)        = x

last [x]          = x
last (_:xs)       = last xs

tail, init        :: [a] -> [a]
tail (_:xs)       = xs

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

null              :: [a] -> Bool
null []           = True
null (_:_)        = False

length            :: [a] -> Int
length            = foldr (const (1+)) 0

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

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

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

iterate           :: (a -> a) -> a -> [a]
iterate f x       =  x : iterate f (f x)

repeat            :: a -> [a]
repeat x          =  xs where xs = x:xs

replicate         :: Int -> a -> [a]
replicate n x     =  take n (repeat x)


cycle             :: [a] -> [a]
cycle []          =  error "Prelude.cycle: empty list"
cycle xs          =  xs' where xs' = xs ++ xs'

tails             :: [a] -> [[a]]
tails xs          =  xs : case xs of
                             []      -> []
                             _ : xs' -> tails xs'
```

```haskell
take, drop             :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs

drop n xs     | n <= 0 =  xs
drop _ []              =  []
drop n (_:xs)          =  drop (n-1) xs

splitAt                :: Int -> [a] -> ([a],[a])
splitAt n xs           =  (take n xs, drop n xs)

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []

dropWhile p []          =  []
dropWhile p xs@(x:xs')
            | p x       =  dropWhile p xs'
            | otherwise =  xs

span :: (a -> Bool) -> [a] -> ([a], [a])
span p as = (takeWhile p as, dropWhile p as)

lines, words     :: String -> [String]
-- lines "apa\nbepa\ncepa\n"
--   == ["apa","bepa","cepa"]
-- words "apa  bepa\n cepa"
--   == ["apa","bepa","cepa"]

unlines, unwords :: [String] -> String
-- unlines ["apa","bepa","cepa"]
--        == "apa\nbepa\ncepa\n"
-- unwords ["apa","bepa","cepa"]
--        == "apa bepa cepa"

reverse          :: [a] -> [a]
reverse          =  foldl (flip (:)) []

and, or          :: [Bool] -> Bool
and              =  foldr (&&) True
or               =  foldr (||) False

any, all         :: (a -> Bool) -> [a] -> Bool
any p            =  or . map p
all p            =  and . map p

elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem x           =  any (== x)
notElem x        =  all (/= x)

lookup      :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []    =  Nothing
lookup key ((x,y):xys)
    | key == x   =  Just y
    | otherwise  =  lookup key xys

sum, product     :: (Num a) => [a] -> a
sum              =  foldl (+) 0
product          =  foldl (*) 1
```

```haskell
maximum, minimum :: (Ord a) => [a] -> a
maximum [] = error "Prelude.maximum: empty list"
maximum (x:xs) = foldl max x xs

minimum [] = error "Prelude.minimum: empty list"
minimum (x:xs) = foldl min x xs

zip              :: [a] -> [b] -> [(a,b)]
zip              =  zipWith (,)

zipWith          :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)
                 =  z a b : zipWith z as bs
zipWith _ _ _    =  []

unzip            :: [(a,b)] -> ([a],[b])
unzip            =
  foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

nub              :: Eq a => [a] -> [a]
nub []           = []
nub (x:xs)       =
           x : nub [ y | y <- xs, x /= y ]

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

(\\)             :: Eq a => [a] -> [a] -> [a]
(\\)             = foldl (flip delete)

union            :: Eq a => [a] -> [a] -> [a]
union xs ys      = xs ++ (ys \\ xs)

intersect        :: Eq a => [a] -> [a] -> [a]
intersect xs ys  = [ x | x <- xs, x `elem` ys ]

intersperse      :: a -> [a] -> [a]
-- intersperse 0 [1,2,3,4] == [1,0,2,0,3,0,4]

transpose        :: [[a]] -> [[a]]
-- transpose [[1,2,3],[4,5,6]]
--    == [[1,4],[2,5],[3,6]]

partition   :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs =
    (filter p xs, filter (not . p) xs)

group            :: Eq a => [a] -> [[a]]
group = groupBy (==)

groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy _  []     =  []
groupBy eq (x:xs) =  (x:ys) : groupBy eq zs
               where (ys,zs) = span (eq x) xs

isPrefixOf :: Eq a => [a] -> [a] -> Bool
isPrefixOf []     _      =  True
isPrefixOf _      []     =  False
isPrefixOf (x:xs) (y:ys) =  x == y
                            && isPrefixOf xs ys
isSuffixOf :: Eq a => [a] -> [a] -> Bool
isSuffixOf x y = reverse x
                   `isPrefixOf` reverse y
```

```haskell
sort               :: (Ord a) => [a] -> [a]
sort               = foldr insert []

insert             :: (Ord a) => a -> [a] -> [a]
insert x []        = [x]
insert x (y:xs)    =
    if x <= y then x:y:xs else y:insert x xs

------------------------------------------
-- * Functions on Char
type String = [Char]

toUpper, toLower :: Char -> Char
-- toUpper 'a' == 'A'
-- toLower 'Z' == 'z'

digitToInt :: Char -> Int
-- digitToInt '8' == 8

intToDigit :: Int -> Char
-- intToDigit 3 == '3'

ord :: Char -> Int
chr :: Int  -> Char
------------------------------------------
-- * Useful functions from Test.QuickCheck

arbitrary :: Arbitrary a => Gen a
-- the generator for values of a type
-- in class Arbitrary, used by quickCheck

choose :: Random a => (a, a) -> Gen a
-- Generates a random element in the given
-- inclusive range.

oneof :: [Gen a] -> Gen a
-- Randomly uses one of the given generators

frequency :: [(Int, Gen a)] -> Gen a
-- Chooses from list of generators with
-- weighted random distribution.

elements :: [a] -> Gen a
-- Generates one of the given values.

listOf :: Gen a -> Gen [a]
-- Generates a list of random length.

vectorOf :: Int -> Gen a -> Gen [a]
-- Generates a list of the given length.

sized :: (Int -> Gen a) -> Gen a
-- construct generators that depend on
-- the size parameter.
------------------------------------------
-- * Useful IO function
putStr, putStrLn  :: String -> IO ()
getLine           :: IO String

type FilePath = String
readFile     :: FilePath -> IO String
writeFile    :: FilePath -> String -> IO ()
```