Re-exam – Introduction to Functional Programming

TDA555/DIT441 (DIT440), HT-23
Chalmers and Göteborgs Universitet, CSE

Day: 2024-01-03, *Time:* 14:00-18:00, *Place:* Johanneberg

Course responsible

Alex Gerdes (072-9744966). He will visit the exam room once around 15:30, 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 definition:

```
f x y
  | y == 0    = x
  | otherwise = f y (x `mod` y)
```

a) What does the expression

```
f 4 10
```

evaluate to? Use equational reasoning and write down the intermediate steps of the evaluation. Note, the `mod` function returns the remainder of an integer division.

b) What is the type signature of `f`?

Solution

```
{-
  f 4 10
= { def f, otherwise case }
  f 10 (4 `mod` 10)
= { apply mod }
  f 10 4
= { def f, otherwise case }
  f 4 (10 `mod` 4)
= { apply mod }
  f 4 2
= { def f, otherwise case }
  f 2 (4 `mod` 2)
= { apply mod }
  f 2 0
= { def f, y == 0 case }
  2
-}

f :: Integral a => a -> a -> a  -- also accepted: Int -> Int -> Int
```

2

Consider the following data type for the colors of the Dutch flag:

```
data Color = Red | White | Blue deriving Show
```

Write a function that partitions a list of colors into separate lists for each color:

```
partitionColors :: [Color] -> ([Color], [Color], [Color])
```

For example:

```
ghci> partitionColors [Red, Blue, White, Red, Blue]
([Red,Red],[White],[Blue,Blue])
```

Note that the `Color` data type does not have an `Eq` instance. You should therefore use a `case`-expression to implement the function. *Hint*: introduce a help function with accumulating parameters for storing the separate color lists.

```
partitionColors = go [] [] []
 where
  go r w b []     = (r, w, b)
  go r w b (x:xs) = case x of
    Red   -> go (x:r) w b xs
    White -> go r (x:w) b xs
    Blue  -> go r w (x:b) xs
```

3

This assignment is about *modeling*. You are going to model a new kind of Social Media in Haskell using data types. We will call it: FPBOOK! FPBOOK is going to be the number one place for functional programmers to meet and discuss topics on FP, such as currying and lazy evaluation.

FPBOOK consists of a list of users and a collection of posts. For every user we store their name, a list of friends, and whether or not (s)he likes functional programming. A post contains a message (modeled as a `String`), the user who posted the message, its visibility, and a list of users that have liked the post. The visibility of a post can be: private, friends, or public.

So, define a collection of data types that models FPBOOK according to the above description.

Suggested solution

Note, you don't need to use record syntax (which the suggested solution uses), nor have deriving `Show`.

```
data FpBook = FpBook
  { users      :: [User]
  , posts      :: [Post]
  }

data User = User
  { name    :: String
  , friends :: [User]  -- or an ID
  , likesFP :: Bool
  }

data Post = Post
  { by      :: User
  , content :: String
  , vis     :: Visibility
  , likes   :: [User]
  }

data Visibility = Private | Friends | Public
```

4

Now that the creation of FPBOOK (see the previous assignment) has started we want to support the developers by writing some code. You are going to define a function:

```
readPost :: IO (String, String, Int)
```

that allows a user to post a message. This `IO` function should ask the user for its name, the to be posted message, and its visibility. We use an integer for denoting the visibility: a '0' stands for private, a '1' for friends, and a '2' for public visibility. The function should check that the integer given by the user is indeed one of these three numbers. If this is not the case, the function should report an error message and exit, that is, it should call the `error` function.

The three parts (user name, message, and visibility) given by the user are then returned in a three-tuple. For example:

```
ghci> readPost
Welcome to FpBook!

What is your name?
> Alex
Please enter your message:
> FP for the win!
Who can see the post? (0 = private, 1 = friends, 2 = public)
> 2
("Alex","FP for the win!",2)
```

Suggested solution

```
readPost = do
  putStr "Welcome to FpBook!\n\nWhat is your name?\n> "
  user <- getLine
  putStr "Please enter your message:\n> "
  msg  <- getLine
  putStr "Who can see the post? (0 = private, 1 = friends, 2 = public)\n> "
  vis  <- readLn
  if vis >= 0 && vis <= 2
    then return (user, msg, vis)
    else error "Invalid visibility!"
```

5

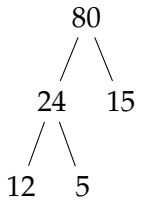
Consider the following data type definition for *non-empty* binary trees:

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

which stores a collection of integers. For example:

```
t :: Tree
t = Node (Node (Leaf 12) 24 (Leaf 5)) 80 (Leaf 15)
```

Which can be depicted as follows:



Your tasks are:

a) Write a function that counts the number of elements in a tree:

```
size :: Tree -> Int
size (Leaf _)     = 1
size (Node l _ r) = size l + 1 + size r
```

b) Define a function that returns the *smallest* element in a tree:

```
minTree :: Tree -> Int
minTree (Leaf x)     = x
minTree (Node l x r) = minimum [x, minTree l, minTree r]
```

Calling the function `minTree` on the example tree `t` will result in:

```
ghci> minTree t
5
```

Note that the trees do not necessarily have the binary search tree property, as the example tree `t` illustrates.

6

In the previous assignment we introduced the `size` and `minTree` functions. We want to make sure that these functions work as aspected by defining some QuickCheck properties. You can do this assignment even if you have not solved the previous assignment.

You may assume that the following function exists:

```
toList :: Tree -> [Int]
toList (Leaf x)     = [x]
toList (Node l x r) = toList l ++ [x] ++ toList r
```

which returns all elements from a tree in a list. You don't need to implement this function, you can just use it.

Your tasks are:

a) Write a property that validates that the `size` function calculates the correct number of elements:

```
prop_size :: Tree -> Bool
prop_size t = size t == length (toList t)
```

b) Implement a property that checks that the `minTree` function indeed returns the smallest element present in a tree:

```
prop_min :: Tree -> Bool
prop_min t = minTree t == minimum (toList t)
```

7

Consider the following data type for polymorphic lists of *even* size:

```
data EvenList a = Nil | Add a (EvenList a) a deriving Show
```

We can only define lists of even size, since the `Add` constructor takes *two* elements of type `a`, and another list of even size. So, the `Add` constructor always increases the size of the list by two, and the size of an empty list is zero (which is an even number).

For example, we can create the following list:

```
elist :: EvenList Int
elist = Add 3 (Add 2 (Add 1 Nil 10) 20) 30
```

Your task is to define a higher-order function:

```
mapEven :: (a -> b) -> EvenList a -> EvenList b
mapEven f list = case list of
  Nil       -> Nil
  Add x l y -> Add (f x) (mapEven f l) (f y)
```

which applies a given function (of type `a -> b`) to every element in a value of type `EvenList a`. We can, for instance, use the `mapEven` function to increment all integers in the example `elist` by one:

```
ghci> mapEven (+1) elist
Add 4 (Add 3 (Add 2 Nil 11) 21) 31
```

Part II

8

Recall the data type for list with an even number of elements from the previous assignment:

```
data EvenList a = Nil | Add a (EvenList a) a
```

We need some more useful functions on this data type. Your tasks are:

a) Write a higher-order function that folds over a value of the `EvenList` data type:

```
foldrEven :: (a -> b -> a -> b) -> b -> EvenList a -> b
foldrEven f b Nil         = b
foldrEven f b (Add x l y) = f x (foldrEven f b l) y
```

The function `foldrEven` behaves the same as `foldr` for normal lists. The function has the following arguments:

- a function that combines two elements of type `a` and the accumulated result of type `b`,
- a base value of type `b`,
- and an `EvenList a`.

We can, for example, use this function to convert an `EvenList` into a normal list:

```
ghci> foldrEven (\ x l y -> x:y:l) [] elist
[3,30,2,20,1,10]
```

b) We want to show values of type `EvenList a` as a proper list, unlike `deriving Show` that shows a bunch of constructors. Suppose we had defined such a function as follows:

```
showEven :: Show a => EvenList a -> String
showEven l = "[" ++ go l ++ "]"
 where
  go Nil         = ""
  go (Add x l y) = show x ++ "," ++ show y ++ "," ++ go l
```

This function has two flaws. The first flaw is that it inserts an unwanted comma after the last element in the list, just before the closing square bracket:

```
ghci> showEven elist
"[3,30,2,20,1,10,]"
```

The second flaw is that it makes excessive use of the append operator `(++)`, which makes the function really inefficient. Your task is to fix both of these flaws. To help you on your way we give some helpful definitions that you *must* use:

```
type StringF = String -> String

charF :: Char -> StringF
charF c = \s -> c:s

showF :: Show a => a -> StringF
showF x = \s -> show x ++ s
```

Instead of using a string directly we are going to represent a string as a *function* that takes a string as argument and produces a string. We use a type synonym `StringF` for this (the 'F' stands for fast). This enables us to use function composition `(.)` to append strings! The `charF` and `showF` are used to convert characters and (small) strings to the fast `StringF` representation. To convert a value of type `StringF` to a normal string you can just apply it to an empty string `""`.

Use the fast string idea to implement the following function:

```
showEvenF :: Show a => EvenList a -> StringF
showEvenF l = charF '[' . go l . charF ']'
 where
  go Nil           = id
  go (Add x Nil y) = showF x . charF ',' . showF y
  go (Add x l   y) = showF x . charF ',' . showF y . charF ',' . go l

--instance Show a => Show (EvenList a) where
--  show = flip showEvenF ""
```

Note that defining `showEvenF` to be equal to `showF` will not solve the problem: this will not produce a proper list (using square brackets and comma separated values).

9

In computer science we often need to *parse* data before we can work with it. When we, for example, read data from a file or receive input via the internet, it will very often be a string of characters. We need to convert the string to a value of a particular data type before we can work with it. This process is called parsing.

A parser is a *function* that takes a string as input and produces output of a particular type:

```
type Parser a = String -> Maybe (a, String)
```

We use the `Maybe` data type to indicate that parsing may fail. Furthermore, we may not use (consume) all characters of the input string, and we return the remaining part of the input string as well. We use this to compose parsers: a parser consumes as much of the input as needed and leaves the remaining part of the input string for the next parser.

A common way to construct parsers is to use a so-called parser combinator library. The following collection of functions is such a library (though quite small):

```
-- Apply a function (type a -> b) to the result of another parser (type a)
app :: Parser (a -> b) -> Parser a -> Parser b

-- Make a parser that returns the given value and doesn't consume any input
ret :: a -> Parser a

-- Left-biased or: try the first and only if it fails consider the second
lor :: Parser a -> Parser a -> Parser a

-- Create a parser for a character, if the predicate holds for that character
sat :: (Char -> Bool) -> Parser Char

-- Apply a parser zero or more times (many) and return the results in a list;
-- many1 applies a parser one or more times.
many, many1 :: Parser a -> Parser [a]

-- Run a parser on a string and return the result
run :: Parser a -> String -> Maybe a
```

With these few functions (also called combinators) we can create very powerful parsers. You don't need the definition for these functions, but they can be found in Appendix A. For example, the following parser recognizes a natural number:

```
pNat :: Parser Int
pNat = ret read `app` many1 (sat isDigit)
```

We try to recognize one or more characters that are digits, which results in a list of digit characters, to which we apply the `read` function, resulting in an `Int`.

If we apply this parser on the string ”123+321” we get:

```
ghci> pNat "123+321"
Just (123,"+321")
```

As you can see, the parser only consumes the first natural number from the input string, and returns the remainder.

Your tasks are:

a) Write a parser combinator (that is, a parser that takes another parser as argument) that parses one `open` character, then the parses `p` followed by parsing one `close` character:

```
within :: (Char, Char) -> Parser a -> Parser a
within (open, close) p = fix `app` char open `app` p `app` char close
 where
  char = \c -> sat (== c)
  fix  = ret $ \ _ x _ -> x
```

For example, using the `within` parser combinator we can recognize a natural number written between angled brackets:

```
ghci> run (within ('<', '>') pNat) "<123>"
Just 123
```

The angled brackets are recognized, but are stripped away and do not end up in the result, which is just a natural number in this example.

b) Next, define a parser combinator that parses a one or more occurrences of a parser `p` separated by a parser `sep`:

```
sepby1 :: Parser a -> Parser b -> Parser [a]
sepby1 p sep = ret (:) `app` p `app` many (ret (\_ x -> x) `app` sep `app` p)
```

We can, for instance, use this very useful parser combinator to recognize a comma separated list of values (natural numbers in this case):

```
ghci> run (pNat `sepby1` sat (== ',')) "1,2,3,42"
Just [1,2,3,42]
```

The comma’s are parsed, but do not end up in the result, which is a list of natural numbers.

A

```haskell
app pf p = \inp -> do
  (f, xs) <- pf inp
  (x, ys) <- p xs
  return (f x, ys)

ret x = \inp -> Just (x, inp)

lor p1 p2 = \inp -> maybe (p2 inp) Just (p1 inp)

sat p (c:cs) | p c = Just (c, cs)
sat _ _            = Nothing

many  p = (ret (:) `app` p `app` many p) `lor` ret []
many1 p = ret (:) `app` p `app` many p

run p = fmap fst . p
```

```haskell
{- Some standard functions from Prelude Data.List
   Data.Maybe Data.Char Control.Monad -}
-----------------------------------------------
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, 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,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, sin, cos, tan :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round, 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]
  -- eval actions from left to right, return results
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
  -- apply action to every element in a list
replicateM :: Monad m => Int -> m a -> m [a]
  -- performs action n times, and return a list

-- 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 Bool ------------------------
(&&), (||)       :: 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 ---------------------------
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)

-- 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]         = []
```

```haskell
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 "cycle: empty list"
cycle xs          =  xs' where xs' = xs ++ xs'

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

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 (x:xs) | p x        = dropWhile p xs
                   | otherwise  = x: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 ["ap","bep","cep"] == "ap\nbep\ncep"
-- unwords ["ap","bep","cep"] == "ap bep cep"
```

```haskell
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

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])
```

```haskell
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

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]

isSpace, isDigit, isAlpha :: Char -> Bool
toUpper, toLower :: Char -> Char

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

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

ord :: Char -> Int
chr :: Int  -> Char

-- functions from Test.QuickCheck ---------------
arbitrary :: Arbitrary a => Gen a
-- generator used by quickCheck

choose :: Random a => (a, a) -> Gen a
-- 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 weighted list of generators

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 a size param

-- IO functions ---------------------------------
putStr, putStrLn :: String -> IO ()
getLine :: IO String
readLn :: Read a => IO a

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