Main topics of the course of Functional Languages

Chapter 7: Higher-order functions

Note: for some working examples, see code in `Examples/Higher-orderFunctions`.

Foldl

The functions `foldl` takes the second argument (initially `v`) and the first item `x` of the list and applies the function `f` to them, then feeds the function with this result and the second item and so on.

```
{- Examples: 1) Input  -> foldl (/) 64 [4, 2, 4]
                Output -> 2.0
             2) Input  -> foldl (\x y -> 2 * x + y) 4 [1, 2, 3]
                Output -> 43
-}

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

Foldr

The function `foldr` takes the second argument (initially `v`) and the last item of the list and applies the function `f`, then it takes the penultimate item from the end and the result, and so on.

```
{- Examples: 1) Input  -> foldr (/) 2 [8, 12, 24, 4]
                Output -> 8.0
             2) Input  -> foldr (\x y -> (x + y) / 2) 54 [12, 4, 10, 6]
                Output -> 12.0
-}

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

Examples

Reverse

```
reverseL :: [a] -> [a]
reverseL = foldl (\xs x -> x:xs) []

reverseR :: [a] -> [a]
reverseR = foldr (\x xs -> xs ++ [x]) []
```

Append

```haskell
appendL :: [a] -> [a] -> [a]
appendL = foldl (\xs x -> xs ++ [x])

appendR :: [a] -> [a] -> [a]
appendR zs ys = foldr (\x xs -> x:xs) ys zs -- = flip (foldr (:))
```

Last

```haskell
lastL :: [a] -> a
lastL (y:ys) = foldl (\xs x -> x) y ys

lastR :: [a] -> a
lastR (y:ys) = foldr (\x xs -> x) y (reverse ys)
```

Flatten

```haskell
flattenR :: [[a]] -> [a]
flattenR = foldr (++) []
```

Length

```haskell
lengthR :: [a] -> Int
lengthR = foldr (\x v -> v + 1) 0

lengthL :: [a] -> Int
lengthL = foldl (\v x -> v + 1) 0
```

Exercises

7.3

Redefinition of `map` and `filter` using `foldr` and `foldl`, given the definitions of the functions `map` and `filter` as follows:

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

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

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

mapL :: (a -> b) -> [a] -> [b]
mapL  = foldl (\xs x -> xs ++ [f x]) []
```

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

filterL :: (a -> Bool) -> [a] -> [a]
filterL p = foldl (\xs x -> if p x then xs ++ [x] else xs) []
```

7.4

Using `foldl`, define a function `dec2int :: [Int] -> Int` that converts a decimal number into an integer.

```haskell
-- Example: dec2int [2,3,4,5], outputs 2345
dec2int :: [Int] -> Int
dec2int = foldl (\v x -> v * 10 + x) 0
```

Chapter 12: Functors, Applicatives and Monads (FAM)

Note: for further information, see code in `Examples/FunctorsApplicativesMonads`.

Making data types into instances of FAM

Observe the following explanatory definition of FAM.

```haskell
instance Functor (Maybe) where
    -- fmap :: (a -> b) -> f a -> f b
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)

instance Applicative (Maybe) where
    -- pure :: a -> f a
    -- pure :: a -> Maybe a
    pure = Just

    -- <*> :: f (a -> b) -> f a -> f b
    -- <*> :: Maybe (a -> b) -> Maybe a -> Maybe b
    Nothing  <*> _  = Nothing
    (Just g) <*> mx = fmap f mx

instance Monad (Maybe) where
    -- return :: a -> f a
    -- return :: a -> Maybe a
    return = pure

    -- (>>=) :: f a -> (a -> f b) -> f b
    -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing  >>= _ = Nothing
    (Just x) >>= g = g xs
```

Note that:

1. **Functors** abstract the idea of mapping a function over each element of a structure.

2. **Applicatives** are Functors that allow functions with any number of arguments to be mapped. Then:
    - The function `pure` converts a value of type `a` into a structure of type `f a`
    - The operator `<*>` is a generalised form of function application for which the argument function, the argument value, and the result value are all contained in `f` structures

    Applicatives can also be viewed as abstracting the idea of applying pure functions to effectful arguments (i.e. arguments are no longer just plain values but may also have effects).
3. **Monads** give back the possibility to fail after that the Applicative style restricted us to applying pure functions to effectful arguments.

Examples

Note: for some working examples, see the code in `Examples/FunctorsApplicativesMonads`.

Partially applied function

```
instance Functor ((->) a) where
    -- fmap :: (b -> c) -> (a -> b) -> (a -> c)
    fmap = (.)

instance Applicative ((->) a) where
    -- pure :: b -> (a -> b)
    pure = const

    -- (<*>) :: (a -> (b -> c)) -> (a -> b) -> (a -> c)
    g <*> h = \x -> g x (h x)

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

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

Zip infinite lists

```
newtype MyZipList a = Z [a] deriving Show

app :: MyZipList a -> [a]
app (Z zs) = zs

instance Functor MyZipList where
    -- fmap :: (a -> b) -> MyZipList a -> MyZipList b
    fmap f (Z zs) = Z (f <$> zs)

instance Applicative MyZipList where
    -- pure a -> MyZipList a
    pure z = Z (repeat z)

    -- (<*>) :: MyZipList (a -> b) -> MyZipList a -> MyZipList b
    Z fs <*> Z zs = Z (fs <*> zs)   -- Z fs <*> Z zs = Z [f z | (f, z) <- zip fs
zs]

instance Monad MyZipList where
```

```
    -- return :: a -> MyZipList a
    return = pure

    -- (>>=) :: MyZipList a -> (a -> MyZipList b) -> MyZipList b
    Z zs >>= f = Z (concat [app (f z) | z <- zs])
```

Expressions

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

instance Functor Expr where
    -- fmap :: (a -> b) -> Expr a -> Expr b
    fmap _ Val         = Val
    fmap f (Var x)     = Var (f x)
    fmap f (Add e1 e2) = Add (fmap f e1) (fmap f e2)

instance Applicative Expr where
    -- pure :: a -> Expr a
    pure = Var

    -- (<*>) :: Expr (a -> b) -> Expr a -> Expr b
    Val   <*> _     = Val
    Var f <*> e     = f <$> e
    Add f1 f2 <*> e = Add (f1 <*> e) (f2 <*> e)

instance Monad Expr where
    -- return :: a -> Expr a
    return = pure

    -- (>>=) :: Expr a -> (a -> Expr b) -> Expr b
    Val   >>= _     = Val
    Var x >>= f     = f x
    Add e1 e2 >>= f = Add (e1 >>= f) (e2 >>= f)
```

State transitions

Note: for some working examples, see the code in `Examples/StateMonads`.

```
type State = Int
newtype ST = S (State -> (a, State))

app :: ST a -> State -> (a, State)
app (S st) = st

instance Functor ST where
    -- fmap :: (a -> b) -> ST a -> ST b
    fmap f stx = S (\s -> let (x, s') = app stx s in (f x, s'))
    -- fmap f stx = do x <- stx
    --                 return (f x)

instance Applicative ST where
    -- pure :: a -> ST a
    pure x = S (\s -> (x, s))

    -- <*> :: ST (a -> b) -> ST a -> ST b
    stf <*> stx = S (\s -> let (f, s')  = app stf s
```

```haskell
                                (x, s'') = app stx s' in (f x, s''))
    -- stf <*> stx = do f <- stf
    --                  x <- stx
    --                  return (f x)

instance Monad ST where
    -- (>>=) :: ST a -> (a -> ST b) -> ST b
    st >>= f = S (\s -> let (x, s') = app st s in app (f x) s')
```

Examples

Relabelling trees

```haskell
-- Tree type with data inside the leaves
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show

-- State transformer
fresh :: ST Int
fresh = S (\n -> (n, n + 1))

-- Applicative style relabelling
relabelA :: Tree a -> ST (Tree Int)
relabelA (Leaf _)   = Leaf <$> fresh
relabelA (Node l r) = Node <$> relabelA l <*> relabelA r

-- Monadic style relabelling
relabelM :: Tree a -> ST (Tree Int)
relabelM (Leaf _)   = do n <- fresh
                         return (Leaf n)
relabelM (Node l r) = do l' <- relabelM l
                         r' <- relabelM r
                         return (Node l' r')
```

Walk the line

Note: for more information look at this example on Learn You a Haskell for Great Good!.

Pierre has decided to take a break from his job at the fish farm and try tightrope walking. He's not that bad at it, but he does have one problem: birds keep landing on his balancing pole! They come and they take a short rest, chat with their avian friends and then take off in search of breadcrumbs. This wouldn't bother him so much if the number of birds on the left side of the pole was always equal to the number of birds on the right side. But sometimes, all the birds decide that they like one side better and so they throw him off balance, which results in an embarrassing tumble for Pierre (he's using a safety net). Let's say that **he keeps his balance if the number of birds on the left side of the pole and on the right side of the pole is strictly lower than 4**. So if there's one bird on the right side and four birds on the left side, he's okay. But if a fifth bird lands on the left side, then he loses his balance and takes a dive. We're going to simulate birds landing on and flying away from the pole and see if Pierre is still at it after a certain number of bird arrivals and departures. For instance, we want to see what happens to Pierre if first one bird arrives on the left side, then four birds occupy the right side and then the bird that was on the left side decides to fly away.

The following types are shared between the next examples and model the number of birds landing on each side of the pole:

```
type Birds = Int
type Pole = (Birds,Birds)
```

Then, without using monadic types we have:

```
landLeft :: Birds -> Pole -> Pole
landLeft n (left,right) = (left + n,right)
```

```
ghci> landLeft  2    (0,0)                          -- (2,0)
ghci> landRight 1    (1,2)                          -- (1,3)
ghci> landRight (-1) (1,2)                          -- (1,1)
ghci> landLeft 2 (landRight 1 (landLeft 1 (0,0)))   -- (3,1)

-- Wrong behaviour!
-- There are 4 birds on the right side and no birds on the left at the same time
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2) --
(0,2)
```

That is, without monadic types we do not model correctly the possible failure of the execution. Thus, using monadic types (i.e. `Maybe`) we have:

```
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left,right)  | abs ((left + n) - right) < 4 = Just (left + n, right)

                         | otherwise                    = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left,right) | abs (left - (right + n)) < 4 = Just (left, right + n)

                         | otherwise                    = Nothing

banana :: Pole -> Maybe Pole
banana _ = Nothing
```

```
ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2  -- Just (2,4)


-- Correct behaviour!
ghci> return (0,0) >>= landLeft 1 >>= landRight 4 >>= landLeft (-1) >>=
landRight (-2)
-- Nothing
```

Using state transition we can achieve an even more descriptive execution:

```
newtype Line a = L (Pole -> (Maybe a, Pole))

app :: Line a -> Pole -> (Maybe a, Pole)
app (L l) = l

instance Functor Line where
    -- fmap :: (a -> b) -> Line a -> Line b
    -- fmap f l = L (\p -> let (mx, p') = app l p in
    --                     case mx of Nothing -> (Nothing, p')
```

```
    --                                  Just x  -> (Just (f x), p'))
    fmap f lx = do x <- lx
                   return (f x)

instance Applicative Line where
    -- pure :: a -> Line a
    pure x = L (\p -> (Just x, p))

    -- (<*>) :: Line (a -> b) -> Line a -> Line b
    -- lf <*> lx = L (\p -> let (mf, p') = app lf p in
    --                      case mf of Nothing -> (Nothing, p')
    --                                 Just f  -> app (f <$> lx) p')
    lf <*> lx = do f <- lf
                   x <- lx
                   return (f x)

instance Monad Line where
    -- return :: a -> Line a
    return = pure

    -- (>>=) :: Line a -> (a -> Line b) -> Line b
    lx >>= f = L (\p -> let (mx, p') = app lx p in
                        case mx of Nothing -> (Nothing, p')
                                   Just x  -> app (f x) p')

landLeft :: Int -> Line ()
landLeft n = L (\p@(l, r) -> if abs (l + n - r) < 4 then (Just (), (l + n, r))
                                                    else (Nothing, p))

landRight :: Int -> Line ()
landRight n = L (\p@(l, r) -> if abs (r + n - l) < 4 then (Just (), (l, r + n))
                                                     else (Nothing, p))

g :: Line ()                  --    (Just () (0, 0))
g = do landLeft     2         -- -> (Just () (2, 0))
       landRight    4         -- -> (Just () (2, 4))
       landLeft     (-1)      -- -> (Just () (1, 4))
       landRight    1         -- -> (Nothing () (1, 4))

execute :: (Maybe (), Pole)
execute = app g (0, 0)
```

Balanced parentheses

```
module BalancedParentheses where
import Control.Applicative

type Stack = String
newtype PDA a = P (Stack -> [(a, Stack)])

app :: PDA a -> Stack -> [(a, Stack)]
app (P p) = p

instance Functor PDA where
    -- fmap :: (a -> b) -> PDA a -> PDA b
    -- fmap f (P px) = P (\s -> case px s of []        -> []
    --                                      [(x, s')] -> [(f x, s')])
```

```
    fmap f pdax = do x <- pdax
                     return (f x)

instance Applicative PDA where
    -- pure :: a -> PDA a
    pure x = P (\s -> [(x, s)])

    -- (<*>) :: PDA (a -> b) -> PDA a -> PDA b
    -- (P pf) <*> pdax = P (\s -> case pf s of
    --                                []        -> []
    --                                [(f, s')] -> let (P x') = f <$> pdax in x'
s')
    pdaf <*> pdax = do f <- pdaf
                       x <- pdax
                       return (f x)

instance Monad PDA where
    -- return :: a -> PDA a
    return = pure

    -- (>>=) :: PDA a -> (a -> PDA b) -> PDA b
    (P px) >>= f = P (\s -> case px s of []        -> []
                                         [(x, s')] -> app (f x) s')

pop :: PDA Char
pop = P (\s -> case s of []     -> []
                         (x:xs) -> [(x, xs)])

push :: PDA ()
push = P (\s -> [((), '(':s)])

balance :: String -> PDA Bool
balance []       = P (\s -> case s of [] -> [(True, s)]
                                      _  -> [(False, s)])
balance ('(':xs) = do {push; balance xs}
balance (')':xs) = do {pop; balance xs}
balance (_:xs)   = balance xs

{-
   Examples: 1) "(1 (2 (3) 4) 5)" -> "Balanced!"
             2) "(1 (2 (3  4) 5)" -> "Error: Too many opened parentheses"
             3) " 1 (2 (3) 4) 5)" -> "Error: Too many closed parentheses"
-}
execute :: String -> IO ()
execute xs = case app (balance xs) [] of
             []             -> print (xs ++ " -> Error: Too many closed
parentheses")
             [(False, s)]   -> print (xs ++ " -> Error: Too many opened
parentheses")
             [(True, [])]   -> print (xs ++ " -> Balanced!")
             _              -> print (xs ++ " -> Fatal error!")
```

Generic functions

An important benefit of abstracting out the concept of Monads is the ability to define generic functions that can be used with any Monad.

```haskell
-- Monadic map, gives the input function f the possibility to fail
mapM :: Monad m  => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)

-- Monadic filter
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filterM p xs
                      return (if b then x:ys else ys)

-- Monadic concat
join :: Monad m => m (m a) -> m a
join mmx = do mx <- mmx
              x  <- mx
              return x
```

Chapter 15: Lazy evaluation

Evaluation strategies

A **redex** is an expression that has the form of a function applied to one or more arguments that can be "reduced" by performing the application. Then, when evaluating an expression, in what order should the reductions be performed? There are two main strategies:

1. **Innermost evaluation**: always choose a redex that is innermost, in the sense that it contains no other redex. If there is more than one innermost redex, by convention we choose the one that begins at the leftmost position in the expression. Using innermost evaluation ensures that arguments are always fully evaluated before functions are applied. That is, arguments are **passed by value**.
2. **Outermost evaluation**: always choose a redex that is outermost, in the sense that it is contained in no other redex. If there is more than one such redex then as previously we choose that which begins at the leftmost position. Using outermost evaluation allows functions to be applied before their arguments are evaluated. That is, arguments are **passed by name**.

In Haskell, **the selection of redexes within the bodies lambda expressions is prohibited**. The rationale for not reducing under lambdas is that functions are viewed as black boxes that we are not permitted to look inside. Using innermost and outermost evaluation, but not within lambda expressions, is normally referred to as **call-by-value** and **call-by-name** evaluation, respectively. Note that **call-by-name may produce a result when call-by-value fails to terminate** (e.g. when dealing with infinite lists). In Haskell, any two different ways of evaluating the same expression will always produce the same final value, provided that they both terminate. However, **call-by-name may require more reduction steps than call-by-value**, in particular when an argument is used more than once in the body of a function. This efficiency problem can be solved by using pointers to indicate sharing of expressions during evaluation. **The use of call-by-name in conjunction with sharing is known as lazy evaluation** (i.e. the default behaviour in Haskell).

Modularization

Modular programming is a software design technique that emphasises separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality. Modularity is considered one of the keys to successful programming. Nevertheless, our ability to effectively decompose a problem into parts depends directly on our ability to glue solutions together. Then, in order to support modular programming, a language must provide good glue. In particular, functional programming languages provide two kinds of glue that the majority of the imperative languages do not support:

1. **Higher-order functions** (such as `foldr` and `foldl`), that capture common recursive patterns and allow **gluing functions together**. This can be achieved because functional languages allow functions that are indivisible in conventional programming languages to be expressed as a combination of parts (i.e. a general higher-order function and some particular specialising functions).
2. **Lazy evaluation**, that allow **gluing programs together**. In particular, lazy evaluation supports function composition in allowing the separation of control from data within the computation. That is, the data is only evaluated as much as required by the control, and the two parts take it in turn to perform reductions (e.g. as when working with infinite lists). Despite Haskell uses lazy evaluation by default, it also provides a special strict version of function application written as `f $! x`. Strict application is mainly used to improve the space performance of programs.

Using these glues, one can modularize programs in new and useful ways. Smaller and more general modules can be reused more widely, easing subsequent programming. This explains why functional programs are so much smaller and easier to write than conventional ones.

Exercises

Note: for further information, see code in `Exercises/chap15`.

15.1

Identify the redexes.

---

1. `1 + (2 * 3)`
    - both: `(2 * 3)`
2. `(1 + 2) * (2 + 3)`
    - innermost 1: `(1 + 2)`
    - innermost 2: `(2 + 3)`
3. `fst (1 + 2, 2 + 3)`
    - outermost: `fst (1 + 2, 2 + 3)`
    - innermost 1: `1 + 2`
    - innermost 2: `2 + 3`
4. `(\x -> 1 + x) (2 * 3)`
    - outermost: `(\x -> 1 + x) (2 * 3)`
    - innermost 1: `(2 * 3)`
    - neither: `1 + x`

15.2

Show why outermost evaluation is preferable to innermost for the purposes of evaluating the expression `fst (1 + 2, 2 + 3)`.

Using an outermost-first strategy, the evaluation is:

- `fst (1 + 2, 2 + 3)`
- `1 + 2`
- `3`

While using an innermost-first strategy, it is:

- `fst (1 + 2, 2 + 3)`
- `fst (3, 2 + 3)`
- `fst (3, 5)`
- `3`

Of course in this case the innermost evaluation policy takes more steps than the outermost.

15.3

Given the definition `mult = \x -> (\y -> x * y)`, show how the evaluation of `mult 3 4` can be broken down into four separate steps.

- `mult 3 4 = (\x -> (\y -> x * y)) 3 4`
- `(\y -> 3 * y) 4`
- `3 * 4`
- `12`

15.4

Generation of the Fibonacci numbers (the sequence such that each number is the sum of the two preceding ones, starting from 0 and 1) using list comprehension.

```haskell
fibs :: [Integer]
fibs = 0 : 1 : [x + y | (x, y) <- zip fibs (tail fibs)]
```

15.5

Appropriate versions of the following functions:

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

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

replicate :: Int -> a -> [a]
replicate n = take n . repeat
```

For the following type of binary trees:

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

```haskell
repeatTree :: a -> Tree a
repeatTree x = t where t = Node t x t

takeTree :: Int -> Tree a -> Tree a
takeTree 0 _            = Leaf
takeTree _ Leaf         = Leaf
takeTree n (Node l x r) = Node (takeTree (n - 1) l) x (takeTree (n - 1) r)

replicateTree :: Int -> a -> Tree a
replicateTree n = takeTree n . repeatTree
```

15.6

Newton's method for computing the square root of a (non-negative) floating-point number `n`.

```haskell
calculateIterations :: Double -> Double -> [(Double, Double)]
calculateIterations s n = [(x, y) | (x, y) <- zip (tail iters) iters]
                          where iters  = iterate next s
                                next a = (a + n / a) / 2

sqroot :: Double -> Double
sqroot n = snd (last (takeWhile f iterations))
           where f (x, y)    = abs (x - y) > 1.0e-5
                 iterations = calculateIterations 0.1 n
```

Chapter 16: Reasoning about programs

Making the append vanish

Many recursive functions are naturally defined using the append operator `++` on lists, but this operator carries a considerable efficiency cost when used recursively. Induction can be used to eliminate such uses of append.

Reverse

```haskell
-- This implementation takes quadratic time in the length of its argument
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

-- This implementation takes linear time in the length of its argument
reverse' :: [a] -> [a] -> [a]
reverse' []     ys = ys
reverse' (x:xs) ys = reverse' xs (x:ys)
```

Note that `reverse'` acts just like `reverse`, specifically `reverse' xs ys = reverse xs ++ ys`.

Base case. `reverse' [] ys = reverse [] ++ ys = ys`

- `reverse' [] ys =`

- `= reverse [] ++ ys =`
- `= [] ++ ys =`
- `= ys`

Inductive case. `reverse' (x:xs) ys = reverse (x:xs) ++ ys`. The inductive hypothesis is `reverse' xs ys = reverse xs ++ ys`.

- `reverse' (x:xs) ys =`
- `= reverse (x:xs) ++ ys =`
- `= reverse xs ++ [x] ++ ys =`
- `= reverse xs ++ ([x] ++ ys) =`
- `= reverse' xs ([x] ++ ys) =`
- `= reverse' xs (x:ys)`

Flatten

```
data Tree = Leaf Int | Node Tree Tree

-- This is inefficient :(
flatten :: Tree -> [Int]
flatten (Leaf n)   = [n]
flatten (Node l r) = flatten l ++ flatten r

-- This is efficient! :)
flatten :: Tree -> [Int] -> [Int]
flatten (Leaf n)   ns = n:ns
flatten (Node l r) ns = flatten' l (flatten' r ns)
```

Note that `flatten'` acts just like `flatten`, specifically `flatten' t ns = flatten t ++ ns`.

Base case:

- `flatten' (Leaf n) ns =`
- `= flatten (Leaf n) ++ ns =`
- `= [n] ++ ns =`
- `= n:ns`

Inductive case:

- `flatten' (Node l r) ns =`
- `= flatten (Node l r) ++ ns =`
- `= flatten l ++ flatten r ++ ns =`
- `= flatten l ++ (flatten r ++ ns) =`
- `= flatten l ++ flatten' r ns =`
- `= flatten' l (flatten' r ns)`

Exercises

Note: for further information, see code in `Exercises/chap16`.

16.1

Show that `add n (Succ m) == Succ (add n m)` by induction on `n`. Remember that:

```
add :: Nat -> Nat -> Nat
add Zero     m = m
add n     Zero = n
add (Succ n) m = Succ (add n m)
```

Base case: `add Zero (Succ m) = Succ (add Zero m)` .

- `add Zero (Succ m) =`
- `= Succ m =`
- `= Succ (add Zero m)`

Inductive case: `add (Succ n) (Succ m) == Succ (add (Succ n) m)`. The inductive hypothesis is `add n (Succ m) == Succ (add n m)`.

- `add (Succ n) (Succ m) =`
- `= Succ (add n (Succ m)) =`
- `= Succ (Succ (add n m)) =`
- `= Succ (add (Succ n) m)`

16.2

Show that addiction is commutative, that is `add n m = add m n` by induction on `n`.

Base case: `add Zero m = add m Zero`.

- `add Zero m =`
- `= m =`
- `= add m Zero`

Inductive case: `add (Succ n) m = add m (Succ n)`. The inductive hypothesis is `add n m = add m n`.

- `add (Succ n) m =`
- `= Succ (add n m) =`
- `= Succ (add m n) =`
- `= add m (Succ n)`

16.3

Using the following definition for the library function that decides if `all` elements of a list satisfy a predicate:

```
all p []     = True
all p (x:xs) = p x && all p xs
```

Show that `replicate` produces a list with identical elements, that is `all (== x) (replicate n x)` by induction on `n >= 0`.

Base case:

- `all (== x) (replicate 0 x) =`

- `= all (== x) [] =  // First case`
- `= True`

Inductive case: the inductive hypothesis is `all (== x) (replicate n x)`.

- `all (== x) (replicate (n + 1) x) =  // Second case`
- `= (x == x) && all (replicate n x) =`
- `= True &&  all (replicate n x) =    // Inductive hypothesis`
- `= True && True =`
- `= True`

16.4

Using the definition:

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

verify the following two properties, by induction on `xs`:

```
xs ++ [] = xs
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```

---

1. **Base case**:
   - `[] ++ [] =    // First case`
   - `= []`

   Inductive case: the inductive hypothesis is `xs ++ [] = xs`.
   - `(x:xs) ++ [] =       // Second case`
   - `= x : (xs ++ []) =   // Inductive hypothesis`
   - `= x:xs`
2. **Base case**:
   - `[] ++ (ys ++ zs) =  // First case`
   - `= (ys ++ zs) =`
   - `= (ys + zs) ++ []`

   Inductive case: the inductive hypothesis is `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.
   - `(x:xs) ++ (ys ++ zs) =      // Second case`
   - `= x : (xs ++ (ys ++ zs)) =  // Inductive hyphotesis`
   - `= x : ((xs ++ ys) ++ zs)`

16.7

Verify the Functor laws for the `Maybe` type, keeping in mind:

```
fmap id      = id                  -- (1)
fmap (g . h) = fmap g . fmap h     -- (2)
```

Remember that:

```haskell
-- fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f Just x  = Just (f x)
fmap f Nothing = Nothing
```

Then:

1. `fmap id = id`.
    - `fmap id Just (x) =`
    - `= Just id x =`
    - `= Just (x)`
2. `fmap (g . h) = fmap g . fmap h`
    - `fmap (g . h) Just (x) =`
    - `= Just ((g . h) x) =`
    - `= Just (g (h x)) =`
    - `= fmap g Just (h x) =`
    - `= fmap g . fmap h`