I/O Exercise 1

In the standard library moduleyou find:

```
data ExitCode = ExitSuccess | ExitFailure Int
              deriving (Eq, Ord, Read, Show)

getArgs :: IO [String]
system  :: String → IO ExitCode
```

Write a program that accepts *name* as argument and then calls LaTeX and BibTeX on *name*.ltx until the document can be expected to stabilise.

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

```
null :: [a] → Bool
null [] = True
null (x : xs) = False

head :: [a] → a
head (x : xs) = x
head [] = error "head: empty list!"

tail :: [a] → [a]
tail (x : xs) = xs
tail [] = error "tail: empty list!"
```

(`head` and `tail` are **partial functions** — both are undefined on the empty list.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

```
length :: [a] → Int
length [] = 0
length (x : xs) = 1 + length xs

(++) :: [a] → [a] → [a]
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

x `elem` [] = False
x `elem` (y : ys)
  = x ≡ y || x `elem` ys
```

```
concat :: [[a]] → [a]
concat [] = []
concat (xs : xss) = xs ++ concat xss

sum [] = 0
sum (x : xs) = x + sum xs

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

(All these functions are in the standard prelude.)

Guarded Definitions

```
sign x | x > 0  = 1
       | x == 0 = 0
       | x < 0  = -1

choose :: Ord a ⇒ (a,b) → (a,b) → b
choose (x,v) (y,w)
    | x > y     = v
    | x < y     = w
    | otherwise = error "I cannot decide!"
```

If no guard succeeds, the next pattern is tried:

```
take_while p (x : xs) | p x = x : take_while p xs
take_while p xs             = []

take_while (< 5) [1, 2, 3]
= take_while (< 5) (1 : 2 : 3 : [])
= 1 : take_while (< 5) (2 : 3 : [])
= 1 : 2 : take_while (< 5) (3 : [])
= 1 : 2 : 3 : take_while (< 5) []
= 1 : 2 : 3 : []
= [1, 2, 3]
```

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

```
take_while p (x : xs) | p x  =  x : take_while p xs
take_while p xs              =  []
```

```
take_while (<5) [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]
= take_while (<5) (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : take_while (<5) (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : take_while (<5) (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : take_while (<5) (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : take_while (<5) (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : take_while (<5) (4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : 4 : take_while (<5) (3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : take_while (<5) (4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : take_while (<5) (5 : 4 : 3 : 4 : 5 : 6 : [])
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : []
= [1, 2, 3, 2, 3, 4, 3, 4]
```


`case` Expressions

```
sign x = case compare x 0 of
           GT ->  1
           EQ ->  0
           LT -> -1
```

The prelude datatype *Ordering* has three elements and is used mostly as result type of the prelude function *compare*:

data *Ordering* = *LT* | *EQ* | *GT*

compare :: *Ord a* ⇒ *a* → *a* → *Ordering*

Another example:

```
choose (x,v) (y,w) = case compare x y of
    GT → v
    LT → w
    EQ → error "I cannot decide!"
```


`if … then … else …` and `case` Expressions

The type *Bool* can be considered as a two-element enumeration type:

data *Bool* = **False** | **True**

Conditional expressions are "syntactic sugar" for **case** expressions over *Bool*:

if *condition* **then** *expr1* **else** *expr2*	≡	**case** *condition* **of** **True** → *expr1* **False** → *expr2*

Two ways of defining functions:

Pattern Matching

```
not True  = False
not False = True
```

`case`

```
not b = case b of
          True  → False
          False → True
```


`case` Expressions are "Anonymous" Pattern Matching

```
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ case xs of
                             [] → []
                             _  → "," : commaWords xs
```

Every use of a `case` expression can be transformed into the use of an auxiliary function defined by pattern matching:

```
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ commaWordsAux xs

commaWordsAux [] = []
commaWordsAux xs = "," : commaWords xs
```

`where` Clauses

If an auxiliary definition is used only locally, it should be inside a **local definition**, e.g.:

```
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ commaWordsAux xs
  where
    commaWordsAux [] = []
    commaWordsAux xs = "," : commaWords xs
```

`where` clauses are visible **only** within their enclosing clause, here "*commaWords* `(`*x* `:` *xs*`) =` …"

`where` clauses are visible within all guards:

```
f x y | y >  z  =  …
      | y == z  =  …
      | y <  z  =  …
  where z = x * x
```


`let` Expressions

Local definitions can also be part of expressions:

```
f k n = let m = k `mod` n
        in if m == 0
           then n
           else f n m

h x y =  let x2 = x * x
             y2 = y * y
         in sqrt (x2 + y2)
```

Definitions can use **pattern bindings**:

```
g k n = let (d,m) = divMod k n
        in if d == 0
           then [m]
           else g d n ++ [m]
```

Guards, `let` and `where` bindings, and `case` cases all are **layout sensitive**!

`let` or `where`?

- `let` *bindings* `in` *expression*

  is an **expression**

- *fname patterns guardedRHSs* `where` *bindings*

  is a clause that is part of a **definition**

- (`where` clauses can also modify `case` cases)

Frequently, the choice between `let` and `where` is a matter of *style*:

- `where` clauses result in a top-down presentation
- `let` expressions lend themselves also to bottom-up presentations

Some Prelude Functions — Elementary List Access

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

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

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

init             :: [a] -> [a]
init [x]          = []
init (x:xs)       = x : init xs

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

Some Prelude Functions — List Indexing

```
length           :: [a] -> Int
length            = foldl' (\n _ -> n + 1) 0

(!!)             :: [b] -> Int -> b
(x:_)  !! 0       = x
(_:xs) !! n | n>0 = xs !! (n-1)
(_:_)  !! _       = error "PreludeList.!!: negative index"
[]     !! _       = error "PreludeList.!!: index too large"
```


Some Prelude Functions — Positional List Splitting

```
take                :: Int -> [a] -> [a]
take 0 _             = []
take _ []            = []
take n (x:xs) | n>0  = x : take (n-1) xs
take _ _             = error "take: negative argument"

drop                :: Int -> [a] -> [a]
drop 0 xs            = xs
drop _ []            = []
drop n (_:xs) | n>0  = drop (n-1) xs
drop _ _             = error "drop: negative argument"

splitAt              :: Int -> [a] -> ([a], [a])
splitAt 0 xs          = ([],xs)
splitAt _ []          = ([],[])
splitAt n (x:xs) | n>0 = (x:xs',xs")
                where (xs',xs") = splitAt (n-1) xs
splitAt _ _       = error "splitAt: negative argument"
```


Some Prelude Functions — Concatenation, Iteration

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

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

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

repeat           :: a -> [a]
repeat x          = xs  where  xs = x:xs
{- repeat x       = x : repeat x -}     -- for understanding

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

cycle            :: [a] -> [a]
cycle xs          = xs'  where  xs' = xs ++ xs'
```


Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                          -- replicate
= take 3 ('!' : repeat '!')                    -- repeat
= '!' : take (3 - 1) (repeat '!')              -- take (iii)
= '!' : take 2 (repeat '!')                    -- subtraction
= '!' : take 2 ('!' : repeat '!')              -- repeat
= '!' : '!' : take (2 - 1) (repeat '!')        -- take (iii)
= '!' : '!' : take 1 (repeat '!')              -- subtraction
= '!' : '!' : take 1 ('!' : repeat '!')        -- repeat
= '!' : '!' : '!' : take (1 - 1) (repeat '!')  -- take (iii)
= '!' : '!' : '!' : take 0 (repeat '!')        -- subtraction
= '!' : '!' : '!' : []                         -- take (i)
= "!!!"
```

Exercise: Splitting with Predicates

- *takeWhile* :: (*a* → *Bool*) → [*a*] → [*a*]

  takeWhile, applied to a predicate *p* and a list *xs*, returns the longest prefix (possibly empty) of *xs* of elements that satisfy *p*.

- *dropWhile* :: (*a* → *Bool*) → [*a*] → [*a*]

  dropWhile p xs returns the suffix remaining after *takeWhile p xs*.

Laws:

- *takeWhile p xs* ++ *dropWhile p xs* = *xs*
- *all p* (*takeWhile p xs*) = **True**
- *null* (*dropWhile p xs*) || (*not* $ *p* $ *head* $ *dropWhile p xs*)

— if *p* is total (on *xs*).

Note: *span p xs* = (*takeWhile p xs*, *dropWhile p xs*)

Some Prelude Functions — List Splitting with Predicates

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

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

span, break      :: (a -> Bool) -> [a] -> ([a],[a])
span p []         = ([],[])
span p xs@(x:xs')
      | p x       = let (ys,zs) = span p xs' in (x:ys,zs)
      | otherwise = ([],xs)

break p           = span (not . p)
```


as-Patterns

```
dropWhile        :: (a -> Bool) -> [a] -> [a]
dropWhile p []    = []
dropWhile p xs@(x:xs')
      | p x       = dropWhile p xs'
      | otherwise = xs
```

Consider matching of the third clause against *dropWhile* (< 5) [1,2,3]:

- *p* = (< 5)
- *xs* = [1,2,3]
- *x* = 1
- *xs'* = [2,3]
- *p x* = (< 5) 1 = 1 < 5 = **True**

Therefore: *dropWhile* (< 5) [1,2,3] = *dropWhile* (< 5) [2,3]

as-Patterns — 2

```
dropWhile        :: (a -> Bool) -> [a] -> [a]
dropWhile p []    = []
dropWhile p xs@(x:xs')
      | p x       = dropWhile p xs'
      | otherwise = xs
```

Consider matching of the third clause against *dropWhile* (< 5) [5,4,3]:

- *p* = (< 5)
- *xs* = [5,4,3]
- *x* = 5
- *xs'* = [4,3]
- *p x* = (< 5) 5 = 5 < 5 = **False**

Therefore: *dropWhile* (< 5) [5,4,3] = [5,4,3]

Some Prelude Functions — List Splitting with Predicates

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

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

span, break      :: (a -> Bool) -> [a] -> ([a],[a])
span p []         = ([],[])
span p xs@(x:xs')
      | p x       = let (ys,zs) = span p xs' in (x:ys,zs)
      | otherwise = ([],xs)

break p           = span (not . p)
```