## Function Application/Types

- Function application is denoted by juxtaposition, and is left associative
- `f x y z` parses as `((f x) y) z`
- If we want `f` applied to both `x`, and the result of the application of `g` to `y`, we must write `f x (g y)`
- In types, the function arrow is right associative
  `Int -> Char -> Bool` parses as `Int -> (Char -> Bool)`
- The type of a function whose first argument is itself a function,
  has to be written as `(a -> b) -> c`
- Note the following types are identical:
  `(a -> b) -> (c -> d)`
  `(a -> b) -> c -> d`

## Sections [*H2010* 3.5]

- A "section" is an operator, with possibly one argument surrounded by parentheses,
  which can be treated as a prefix function name.
- `(+)` is a prefix function adding its arguments
  (e.g. `(+) 2 3  = 5`)
- `(/)` is a prefix function dividing its arguments
  (e.g. `(/) 2.0 4.0  = 0.5`)
- `(/4.0)` is a prefix function dividing its single argument by 4
  (e.g. `(/4.0) 10.0 = 2.5`)
- `(10.0/)` is a prefix function dividing 10 by its single argument
  (e.g. `(10/) 4.0 = 2.5`)
- `(- e)` is not a section, use `subtract e` instead.
  (e.g. `(subtract 1) 4 =  3`)

## Writing Functions (I) — using other functions

(Examples from Chp 4, Programming in Haskell, 2nd Ed., Graham Hutton 2016)

- Function `even` returns true if its integer argument is even

  ```
  even n = n `mod` 2 == 0
  ```

  We use the modulo function `mod` from the Prelude
- Function `recip` calculates the reciprocal of its argument

  ```
  recip n = 1/n
  ```

  We use the division function `/` from the Prelude
- Function call `splitAt n xs` returns two lists, the first with the first `n` elements of `xs`, the second with the rest of the elements

  ```
  splitAt n xs = (take n xs, drop n xs)
  ```

  We use the list functions `take` and `drop` from the Prelude

## Writing Functions (II) — using recursion

- We shall show how to write the functions `take` and `drop` using recursion.
- We shall consider what this means for the execution efficiency of `splitAt`.
- We then do a direct recursive implementation of `splitAt` and compare.

## Implementing take

- `take :: Int -> [a] -> [a]`
  Let `xs1 = take n xs` below.
  Then `xs1` is the first `n` elements of `xs`.
  If `n <= 0` then `xs1 = []`.
  If `n >= length xs` then `xs1 = xs`.
- ```
  take n _ | n <= 0  =  []
  take _ []          =  []
  take n (x:xs)      =  x : take (n-1) xs
  ```
- How long does `take n xs` take to run?
  (we count function calls as a proxy for execution time)
- It takes time proportional to `n` or `length xs`, whichever is shorter.

## Implementing drop

- `drop :: Int -> [a] -> [a]`
  Let `xs2 = drop n xs` below.
  Then `xs2` is `xs` with the first `n` elements removed.
  If `n <= 0` then `xs2 = xs`.
  If `n >= length xs` then `xs2 = []`.
- ```
  drop n xs | n <= 0  =  xs
  drop _ []           =  []
  drop n (x:xs)       =  drop (n-1) xs
  ```
- How long does `drop n xs` take to run?
- It takes time proportional to `n` or `length xs`, whichever is shorter.

## Implementing splitAt recursively

- `splitAt :: Int -> [a] -> ([a],[a])`
  Let `(xs1,xs2) = splitAt n xs` below.
  Then `xs1` is the first `n` elements of `xs`.
  Then `xs2` is `xs` with the first `n` elements removed.
  If `n >= length xs` then `(xs1,xs2) = (xs,[])`.
  If `n <= 0` then `(xs1,xs2) = ([],xs)`.
- ```
  splitAt n xs | n <= 0  =  ([],xs)
  splitAt _ []           =  ([],[])
  splitAt n (x:xs)
    = let (xs1,xs2) = splitAt (n-1) xs
      in  (x:xs1,xs2)
  ```
- How long does `splitAt n xs` take to run?
- It takes time proportional to `n` or `length xs`, whichever is shorter, which is twice as fast as the version using `take` and `drop` explicitly!

## Switcheroo!

- Can we implement `take` and `drop` in terms of `splitAt`?
- Hint: the Prelude provides the following:
  ```
  fst :: (a,b) -> a
  snd :: (a,b) -> b
  ```
- Solution:
  ```
  take n xs = fst (splitAt n xs)
  drop n xs = snd (splitAt n xs)
  ```
- How does the runtime of these definitions compare to the direct recursive ones?
  ```