Contents

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 (n: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 te version using `take` and `drop` explicitly

take and drop

- Can we implement `take` and `drop` in terms of `splitAt`
- 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 firect recursive ones?

Higher Order Functions

What is the difference between these two functions?

```
add x y = x + y
add2 (x, y) = x + y
```

We can see it in the types; `add` is *curried*, takng one argument at a time

```
add :: Integer -> Integer -> Integer
add2 :: (Integer, Integer) -> Integer
```

Any type `a -> a -> a` can also be written `a -> (a -> b)`. The function type arrow associates to the right.

In Haskell, functions are *first class citizens*. In other words, they occupy the same status in the langauge as values: you can pass them as arguments, make them part of data structions, compute them as the result of functions…

```
add3 :: (Integer -> (Integer -> Integer))
add3 :: add

> add3 1 2
3
```

Notice that there are no parameters in the definition of `add3`.

A function with multiple arguments can be viewed as a function of one argument, which computes a new function

```
add 3 4
  ==> (add 3) 4
  ==> ((+) 3) 4
```

The first place you might encounter this is the notion of *partial application*

```
increment :: Integer -> Integer
increment = add 1
```

If the type of `add` is `Integer -> Integer -> Integer`, and the type of `add 1 2` is `Integer`, the the type of `add 1` is?

It is `Integer -> Integer`

Examples of Partial Application

```
second :: [a] -> a
second = head . tail

> second [1, 2, 3]
2
```

An infix operator can be partially applied by taking a section:

```
increment = (1, +) -- or (+, 1)

addnewline = (++"\n")

double :: Integer -> Integer
double = (+2)

> [double x | x <- [1..10] ]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Functions can be taken as parameters as well

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)

addtwo = twice increment
```

This could also be written

```
twice f = f . f
```

Composition

In fact, we can define composition using this technique:

```haskell
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)

twice f = f `compose` f
```

Super-bonus hack. Haskell permits the definition of infix functions:

```haskell
(f ! g) x = f (g x)
twice f = f!f
```

Function composition is in fact part of the Haskell Prelude

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
```