Haskell MOOC

Part 2

by Joel Kaasinen (Nitor) and John Lång (University of Helsinki)

9 Lecture 9: Recap of Part 1

This lecture goes over the basic parts of Haskell introduced in part 1 of the course: types, values, pattern matching, functions and recursion.

9.1 Types

Remember the primitive types of Haskell? Here they are:

Values	Type	Meaning
`True`, `False`	`Bool`	Truth values
`0`, `1`, `20`, `-37`, ...	`Int`	Whole numbers
`'A'`, `'a'`, `'!'`, ...	`Char`	Characters
`""`, `"abcd"`, ...	`String`	Strings, which are actually just lists of characters, `[Char]`
`0.0`, `-3.2`, `12.3`, ...	`Double`	Floating-point numbers
`()`	`()`	The so-called unit type with only one value

It’s possible to combine these primitive types in various ways to form more complex types. Function types, tuple types and list types are examples of types that combine other types.

Values	Type	Meaning
`(1,2)`, `(True,'c')`, ...	`(a, b)`	A pair of a value of type `a` and a value of type `b`
`(1,2,3)`, `(1,2,'c')`, ...	`(a, b, c)`	A triple of values (of types `a`, `b` and `c`)
`[]`, `[1,2,3,4]`, ...	`[a]`	List of values of type `a`
`not`, `reverse`, `\x -> 1`, `\x -> x`, ...	`a -> b`	Function from type `a` to type `b`

There's one more powerful mechanism for creating more types: *Algebraic datatypes* (ADTs). Some examples include:

```haskell
-- Enumeration types
data Bool = True | False
data Color = Red | Green | Blue

-- Record types that contain fields
data Vector2d = MakeVector Double Double
data Person = Person Int String

-- Parameterized types. Note the type parameter `a`
data PairOf a = TwoValues a a

-- Recursive types
data IntList = Empty | Node Int IntList

-- Complex types which combine many of these features
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data List a = Nil | Cons a (List a)             -- This is
        equivalent to the built-in [a] type
data Tree a = Leaf a | Node a (Tree a) (Tree a)
data MultiTree a = MultiTree a [MultiTree a]     -- Note the list
```

Values of these types include:

Values	Type
`True`, `False`	`Bool`
`Red`, `Green`, `Blue`	`Color`
`MakeVector 1.5 3.2`	`Vector2d`
`Person 13 "Bob"`	`Person`
`TwoValues 1 3`	`PairOf Int`
`Empty`, `Node 3 (Node 4 Empty)`	`IntList`
`Nothing`, `Just 3`, `Just 4`, ...	`Maybe Int`
`Nothing`, `Just 'c'`, `Just 'd'`, ...	`Maybe Char`
`Left "foo"`, `Right 13`, ...	`Either String Int`

Values	Type
`Nil`, `Cons True Nil`, `Cons True (Cons False Nil)` ...	`List Bool`
`Leaf 7`, `Node 1 (Leaf 0) (Leaf 2)`, ...	`Tree Int`
`MultiTree 'a' [MultiTree 'b' [], MultiTree 'c' []]`, ...	`MultiTree Char`

You can combine parameterized types in complex ways, for example with something like `Either [String->String] (Maybe String, Int)`.

The names of concrete types start with capital letters. Lowercase letters are used for *type variables* which indicate *parametric polymorphism*: functions and values that can have multiple types. Here are some examples of polymorphic function types:

```
[a] -> [a]    -- function from list of any type, to list of the same
         type
[a] -> a      -- function from list of any type, to the element type
(a,b) -> [a]  -- function from tuple to list
```

9.1.1 More About Lists

List literals can be written in using the familiar `[x,y,z]` syntax. However, that notation is just a shorthand as lists are actually built up of the list constructors `[]` and `(:)`. These constructors are also used when pattern matching lists. Here are some examples of lists:

Abbreviation	Full list	Type
`[1,2,3]`	`1:2:3:[]`	`[Int]`
`[[1],[2],[3]]`	`(1:[]):(2:[]):(3:[]):[]`	`[[Int]]`
`"foo"`	`'f':'o':'o':[]`	`[Char]`, also known as `String`

There's also a range syntax for lists:

Range	Result
`['a' .. 'z']`	`"abcdefghijklmnopqrstuvwxyz"`
`[0 .. 9]`	`[0,1,2,3,4,5,6,7,8,9]`
`[0, 5 .. 25]`	`[0,5,10,15,20,25]`
`[x .. y]`	everything from `x` to `y`
`[9 .. 3]`	`[]`
`[y, y-1 .. x]`	everything from `y` to `x` in decreasing order
`[9,8 .. 3]`	`[9,8,7,6,5,4,3]`

List comprehensions are another powerful way to create lists:

Comprehension	Result
`[x^3 | x <- [1..3]]`	`[1,8,27]`
`[x^2 + y^2 | x <- [1..3], y <- [1..2]]`	`[2,5,5,8,10,13]`
`[y | x <- [1..10], let y = x^2, even x, y<50]`	`[4,16,36]`
`[c | c <- "Hello, World!", elem c ['a'..'z']]`	`"elloorld"`

In general, `[f x | x <- xs, p x]` is the same as `map f (filter p xs)`. Also, `[y | x <- xs, let y = f x]` is the same as `[f x | x <- xs]`. Any combination of `<-`, `let` and `[f x | ...]` is possible.

Just one more note on the syntax. Recall that `(:)` associates to the right, e.g. `True:False:[]` is the same as `True:(False:[])`. (In fact, `(True:False):[]` is not even a list, because `True:False` attempts to add `True` in front of `False` which is not a list.)

9.2 Functions

The basic form of function definition is:

```
functionName :: argumentType -> returnType
functionName argument = returnValue
```

For example:

```
repeatString :: String -> String
repeatString s = s ++ s
```

Functions taking multiple arguments are defined in a similar manner. Note how the type of a multi-argument function looks like.

```
surroundString :: String -> String -> String
surroundString around s = around ++ s ++ around
```

Functions can be polymorphic, can take multiple arguments, and can even take functions as arguments. Here are more examples:

```
id :: a -> a
id x = x

const :: a -> b -> a
const x y = x
```

```haskell
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

More sophisticated functions can be defined using *pattern matching*. We can pattern match on the *constructors* of algebraic datatypes like `Maybe`, and also lists with the constructors `[]` and `(:)`.

```haskell
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)

maybe :: b -> (a -> b) -> Maybe a -> b
maybe def _ Nothing  = def
maybe _   f (Just x) = Just (f x)

safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x
```

Yet more sophistication can be achieved with *guards*. Guards let you define a function case-by-case based on tests of type `Bool`. Guards are useful in situations where pattern matching can' t be used. Of course, guards can also be combined with pattern matching:

```haskell
myAbs :: Int -> Int
myAbs x
  | x < 0     = -x
  | otherwise = x

safeDiv :: Double -> Double -> Maybe Double
safeDiv x y
  | y == 0    = Nothing
  | otherwise = Just (x / y)

buy :: String -> Double -> String
buy "Banana" money
  | money < 3.2   = "You don't have enough money for a banana"
  | otherwise     = "You bought a banana"
buy product  _    = "No such product: " ++ product
```

Case expressions let us pattern match inside functions. They are useful in situations where the result of one function depends on the result of another and we want to match the pattern on the output of the other function:

```haskell
divDefault :: Double -> Double -> Double -> Double
divDefault x y def = case safeDiv x y of
  Nothing -> def
  Just w  -> w
```

Let-expressions enable *local definitions*. Where-clauses work similarly to `lets`. For example:

```haskell
circleArea :: Double -> Double
circleArea r = let pi = 3.1415926
                   square x = x * x
               in pi * square r

circleArea' :: Double -> Double
circleArea' r = pi * square r
    where pi = 3.1415926
          square x = x * x
```

Lambda expressions are another occasionally useful syntax for defining functions. Lambda expressions represent anonymous (unnamed) functions. They can be used for defining local functions that are typically used only once.

```haskell
incrementAll :: [Int] -> [Int]
incrementAll xs = map (\x -> x + 1) xs
```

Note that `f x = y` is the same thing as `f = \x -> y`.

Finally, binary operators have *sections*. Sections are partially applied operators. The section of an operator is obtained by writing the operator and one of its arguments in parentheses. For example, `(*2)` multiplies its argument by `2` from the right, e.g. `(*2) 5 ==> 5 * 2`. A fractional number (e.g. a `Double`) can be inverted with the section `(1/)`, e.g. `(1/) 2 ==> 0.5`.

```haskell
incrementAll' :: [Int] -> [Int]
incrementAll' xs = map (+1) xs
```

9.3 Functional Programming

Haskell is a functional programming language, which means that functions can be passed in as arguments and returned from functions. As a programming paradigm,

functional programming aims to build programs by combining simple functions together to form larger and larger ones.

The most often presented example of functional programming is functional list manipulation using *higher-order functions* (functions that take functions as arguments) like `map` and `filter`. Here's one example from part 1:

```
-- a predicate that checks if a string is a palindrome
palindrome :: String -> Bool
palindrome str = str == reverse str

-- palindromes n takes all numbers from 1 to n, converts them to
-- strings using show, and keeps only palindromes
palindromes :: Int -> [String]
palindromes n = filter palindrome (map show [1..n])
```

```
palindromes 150
  ==> ["1","2","3","4","5","6","7","8","9",
       "11","22","33","44","55","66","77","88","99",
       "101","111","121","131","141"]
```

We also encountered other functional programming patterns in part 1, like *partial application*:

```
map (take 3) [[1,2,3,4,5],[6,7,8,9,0]]
  ==> [[1,2,3],[6,7,8]]
```

Also, *function composition*:

```
(map reverse . filter (/="Smith")) ["Jones","Smith","White"]
  ==> ["senoJ","etihW"]
map (negate . sum) [[1,2,3],[5]]
  ==> [-6,-5]
```

And finally, *folds*:

```
foldr (*) 1 [2,3,4]  ==> 24
foldr max 0 [1,3,7]  ==> 7
foldr (++) "" ["abc","de","f"] ==> "abcdef"
```

9.4 Recursion

To implement a function that uses repetition in Haskell, you need recursion. Haskell has no loops like other programming languages. Here are some simple recursive functions in Haskell:

```haskell
repeatString :: Int -> String -> String
repeatString 0 s = ""
repeatString n s = s ++ repeatString (n-1) s

times :: Int -> Int -> Int
times 0 n = 0
times 1 n = n
times m n = n + times (m - 1) n

safeLast :: [a] -> Maybe a
safeLast []     = Nothing
safeLast [x]    = Just x
safeLast (x:xs) = safeLast xs
```

To consume or produce a list you often need recursion. Here are the implementations of `map` and `filter` as examples of recursive list processing:

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

filter :: (a -> Bool) -> [a] -> [a]
filter _    []     = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      = filter pred xs
```

Sometimes, a recursive helper function is needed in case you need to keep track of multiple pieces of data.

```haskell
sumNumbers :: [Int] -> Int
sumNumbers xs = go 0 xs
  where go sum [] = sum
        go sum (x:xs) = go (sum+x) xs
```

Here’s a final example utilizing guards, pattern matching, a helper function and recursion:

```haskell
-- split a string into pieces at the given character
mySplit :: Char -> String -> [String]
mySplit c xs = helper [] xs
  where helper piece [] = [piece]
        helper piece (y:ys)
          | c == y    = piece : helper [] ys
          | otherwise = helper (piece++[y]) ys



mySplit '-' "a-bcd-ef"  ==>  ["a","bcd","ef"]
```

9.5 Type Classes

The following functions are *parametrically polymorphic*:

```haskell
id :: a -> a
id x = x

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

fst :: (a,b) -> a
fst (x,y) = x
```

Parametrically polymorphic functions always work the same way, no matter what types we're working with. This means that we can't define a special implementation of `id` just for the `Int` type, or `fst` for the type `(Bool, String)`.

By contrast, *ad hoc polymorphism* allows different types to have different implementations of the same function. Ad hoc polymorphism in Haskell can be achieved by defining a *type class* and then declaring *instances* of that type class for various types. Ad hoc polymorphism is a handy way for expressing a common set of operations, even when the implementation of the operations depends on the type they're acting on.

Functions that use ad hoc polymorphism have a *class constraint* in their types. Here are some examples:

```haskell
negate :: Num a => a -> a
(==) :: Eq a => a -> a -> Bool
sort :: Ord a => [a] -> [a]
```

A type like `Num a => a -> a` means: for any type `X` that's a member of the `Num` class, this function has type `X -> X`. In other words, we can invoke `negate` on any number type, but not on other types:

```
Prelude> negate 1
-1
Prelude> negate 1.0
-1.0
Prelude> negate True
<interactive>:3:1: error:
    • No instance for (Num Bool) arising from a use of ‘negate’
```

Here's a summary of some useful type classes from the standard library.

- Comparison
  - `Eq` is for equality comparison. It contains the `==` operator
  - `Ord` is for order comparison. It contains the ordered comparison operators like `<` and `=>`, and functions like `max` and `min`.
- Numbers
  - `Num` is for all number types. It contains `+`, `-`, `*` and `negate`.
  - `Integral` is for whole number types. Most notably, it contains integer division `div`.
  - `Fractional` is for number types that support division, `/`
- Converting values to strings
  - `Show` contains the function `show :: Show a => a -> String` that converts values to strings
  - `Read` contains the function `read :: Read a => String -> a` that is the inverse of `show`

Sometimes, multiple class constraints are needed. For instance here:

```
sumTwoSmallest :: (Num a, Ord a) => [a] -> a
sumTwoSmallest xs = let (a:b:_) = sort xs
                    in a+b
```

Now that we've seen some classes and types, let's look at the syntax of declaring classes and instances. Here are two class definitions:

```
class Sized a where
  empty :: a        -- a thing with size 0
  size :: a -> Int
```

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

Consider the following data structures:

```haskell
data Numbers = None | One Int | Two Int Int
data IntList = Nil | ListNode Int IntList
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

All of these have sizes that we can count, but we need to perform the operation differently:

```haskell
instance Sized Numbers where
  empty = None
  size None      = 0
  size (One _)   = 1
  size (Two _ _) = 2

instance Sized IntList where
  empty = Nil
  size Nil               = 0
  size (ListNode _ list) = 1 + size list

instance Sized (Tree a) where
  empty = Leaf
  size Leaf = 0
  size (Node _ left right) = 1 + size left + size right
```

We can also easily declare `Eq` instances for `Numbers` and `IntList`:

```haskell
instance Eq Numbers where
  None      == None       = True
  (One x)   == (One y)    = x==y
  (Two x y) == (Two z w)  = x==z && y==w
  _         == _          = False         -- to handle cases like
        None == One 1

instance Eq IntList where
  Nil             == Nil               = True
  (ListNode x xs) == (ListNode y ys)   = x == y && xs == ys
  _               == _                 = False
```

However, since the `Tree` datatype is parameterized over the element type `a`, we need an `Eq a` instance in order to have an `Eq (Tree a)` instance. This is achieved by

adding a class constraint to the instance declaration. This is called an *instance hierarchy*.

```
instance Eq a => Eq (Tree a) where
  Leaf         == Leaf               = True
  (Node x l r) == (Node x' l' r')    = x == x' && l == l' && r == r'
  _            == _                  = False
```

9.5.1 Deriving

Some standard type classes, most notably `Show`, `Read`, `Eq` and `Ord` can be *derived*, that is, you can ask the compiler to generate automatic instances for you. For example we could have derived all of these classes for our earlier `Numbers` example.

```
data Numbers = None | One Int | Two Int Int
  deriving (Show, Read, Eq, Ord)
```

```
None == One 1        ==> False
Two 1 2 == Two 1 2   ==> True
None < Two 1 2       ==> True
Two 1 3 < Two 1 2    ==> False
show (Two 1 3)       ==> "Two 1 3"
```

9.6 Quiz

What is the type of `('c',not)`

a. `[Char]`
b. `[Bool]`
c. `(Char,Bool -> Bool)`
d. `(Char,Bool)`
e. It is a type error.

What is the type of `['c',not]`

a. `[Char]`
b. `[Bool]`
c. `(Char,Bool -> Bool)`
d. `(Char,Bool)`
e. It is a type error.

Which of these is a value of the following type?

```
data T = X Int | Y String String | Z T
```

a. `X "foo"`
b. `Y "foo"`
c. `Z (X 1)`
d. `X (Z 1)`

What is the type of this function?

```
f (_:Just x:_) = x
f _            = False
```

a. `Maybe a -> a`
b. `[Maybe a] -> a`
c. `[Maybe a] -> Bool`
d. `[Maybe Bool] -> Bool`

What is the type of this function?

```
f x y = x-y == 0
```

a. `(Num a, Eq a) => a -> a -> Bool`
b. `Num a => a -> a -> Bool`
c. `Eq a => a -> a -> Bool`
d. `a -> a -> Bool`

Which of the following types can `x` have in order for `x (&&) y` to not be a type error?

a. `Bool`
b. `Bool -> Bool -> Bool`
c. `(Bool -> Bool -> Bool) -> Bool -> Bool`

9.7 Working on the Exercises

Here's a short recap of how to work on the exercises. The system is the same as for part 1 of the course.

1. Clone the GitHub repository https://github.com/moocfi/haskell-mooc

2. Go to the `exercises/` directory
3. Run `stack build` to download dependencies
4. Edit the file `Set9a.hs`
5. Check your answers by running `stack runhaskell Set9aTest.hs`
6. Return your answers on the Submit page
7. Repeat as necessary. You can work on the exercise sets in any order, and you can return the sets as many times as you want!
8. You can see total score on the My status page

9.8 Exercises

- Set9a - small exercises that recap part 1 of the course
- Set9b - let's solve the N Queens puzzle!

10 Lecture 10: Reductionism

- Purity
- Laziness
- Haskell evaluation

10.1 Laziness & Purity

Purity and laziness were mentioned as key features of Haskell in the beginning of part 1. Let's take a closer look at them.

Haskell is a *pure* functional language. This means that the value `f x y` is always the same for given `x` and `y`. In other words, the values of `x` and `y` uniquely determine the value of `f x y`. This property is also called *referential transparency*.

Purity also means that there are no side effects: you can't have the evaluation of `f x y` read a line from the user - the line would be different on different invocations of `f` and would affect the return value, breaking referential transparency! Obviously you need side effects to actually get something done. We'll get back to how Haskell handles side effects later.

Haskell is a *lazy* language. This means that a value is not evaluated if it is not needed. An example illustrates this best. Consider these two functions:

```
f x = f x   -- infinite recursion
g x y = x
```

Evaluating `f 1` does not halt due to the infinite recursion. However, this works:

```
g 2 (f 1)  ==>  2
```

Laziness is not a problem because Haskell is pure. Only the result of the function matters, not the side effects. So if the result of a function is not used, we can simply not evaluate it without changing the meaning (semantics) of a program. Well okay, sometimes we get a terminating program instead of one that goes on for ever, but adding laziness never makes a functioning Haskell program break.

If you're interested in the theory behind this, check out the Church-Rosser theorem or the Haskell Wiki article Lazy vs. non-strict.

10.2 Equational Reasoning

Referential transparency, the feature that an expression always returns the same value for the same inputs, is a very powerful property that we can leverage to *reason about programs*.

In a C-style language, we might write a procedure that may not always return the same value for the same arguments:

```
int c = 0;
int funny(int x) {
  return x + c++;
}
```

The expression `c++` increments the value of `c` and returns the old value of `c`. The next time it is evaluated, the value of `c` has increased by one. This means that depending on the current value of `c`, `funny(0)` might return `0`, `1`, `2`, or any other integer value. (It might even return negative values if `c` overflows!)

In some situations this kind of behaviour with side-effects may be useful, but there are also times when it is more important to be able to reason about the code easily. The advantage of pure functions is that they can be analyzed using basic mathematical techniques. Sometimes applying math to our functions can even reveal simplifications or optimisations we otherwise wouldn't have thought about.

Consider the following expression:

```
map (+1) . reverse . map (-1)
```

This expression can be simplified to just `reverse`. We begin by establishing some helpful facts (or *lemmas*). First, suppose that we know

1. `map id          === id`
2. `map f . map g   === map (f.g)`
3. `reverse . map f === map f . reverse`

The fourth fact that we're going to need is the following:

4. `(+1) . (-1) === id`

We can prove fact 4 by reasoning about how `(+1) . (-1)` behaves for an arbitrary input `x`:

```
((+1) . (-1)) x === ((+1) ((-1) x))
                === ((+1) (x - 1))
                === (x - 1) + 1
                === x
                === id x
```

Because we didn't assume anything about `x`, we may conclude that the above chain of equations holds for *every* `x`. Thus,

```
(+1) . (-1) === id
```

For those who are familiar with the technique of *proof by induction*, it is a fun exercise to prove the first three facts also. This course doesn't discuss induction proofs, though, so don't sweat if you don't know induction.

Now, from facts 1-4 it follows that

```
    map (+1) . reverse . map (-1)
=== map (+1) . (reverse . map (-1))    -- By associativity of (.)
=== map (+1) . (map (-1) . reverse)    -- By fact 3
=== (map (+1) . map (-1)) . reverse    -- By associativity of (.)
=== map ((+1) . (-1)) . reverse        -- By fact 2
=== map id . reverse                   -- By fact 4
=== id . reverse                       -- By fact 1
=== reverse                            -- By the definition of id
```

This course won't go into details about proving things about programs, but it's good to know that pure functional programming is very compatible with analysis like this.

10.3 Infinite Lists

The benefits of laziness are best demonstrated with some examples involving *infinite lists*. Let's start with `repeat 1`, which generates an infinite list of `1`s. If we try to tell GHCi to print the value `repeat 1`, it will just keep printing `1`s for ever until we interrupt it using Control-C:

```
Prelude> repeat 1
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
^C
```

However, due to laziness, we can work with infinite lists and write computations that end. We just need to use a finite number of elements from the infinite list. Here are some examples:

```
Prelude> take 10 $ repeat 1
[1,1,1,1,1,1,1,1,1,1]
Prelude> take 20 $ repeat 1
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
Prelude> repeat 1 !! 13337
1
```

An infinite list that just repeats one element can sometimes be necessary, but it's kind of pointless. Let's look at some more useful infinite lists next. You can use the `[n..]` syntax to generate an infinite list of numbers, starting from `n`:

```
Prelude> take 20 [0..]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
Prelude> take 10 . map (2^) $ [0..]
[1,2,4,8,16,32,64,128,256,512]
```

The function `cycle` repeats elements from the given list over and over again. It can be useful when dealing with rotations or cycles.

```
Prelude> take 21 $ cycle "asdf"
"asdfasdfasdfasdfasdfa"
Prelude> take 4 . map (take 4) . tails $ cycle "asdf"
["asdf","sdfa","dfas","fasd"]
```

10.3.1 Example: Transaction Numbers

As a more concrete example of how `cycle` is useful, let's look at computing the check digit of Finnish bank transfer transaction numbers (viitenumero). A transaction

number consists of any number of digits, followed by a single check digit. The check digit is checked by multiplying the digits (from right to left) with the numbers 7, 3, 1, 7, 3, 1 and so on, and summing the results. If the result of the sum *plus the check digit* is divisible by 10, the number is valid.

Here's a concrete example. `116127` is a valid transaction number. The computation goes like this:

```
digits:       1  1  6  1  2
              *  *  *  *  *
multipliers:  3  7  1  3  7
              3+ 7+ 6+ 3+14 = 33
check digit is 7, 33+7=40 is divisible by 10, valid
```

Here's the Haskell code for a transaction number checker. Note how we use use the infinite list `cycle [7,3,1]` for the multipliers.

```haskell
viitenumeroCheck :: [Int] -> Bool
viitenumeroCheck allDigits = mod (checksum+checkDigit) 10 == 0
  where (checkDigit:digits) = reverse allDigits
        multipliers = cycle [7,3,1]
        checksum = sum $ zipWith (*) multipliers digits
```

```haskell
viitenumeroCheck [1,1,6,1,2,7]  ==> True
viitenumeroCheck [1,1,6,1,2,8]  ==> False
```

10.3.2 Example: Finding a Power

Finally, here's how you would find the first power of 3 that's larger than 100.

```haskell
Prelude> head . filter (>100) $ map (3^) [0..]
243
```

Let's go through how this works step by step. Note how map and filter are processing the list lazily, one element at a time, as needed. This is similar to how *generators* or *iterators* work in languages like Python or Java.

```haskell
    head (filter (>100) (map (3^) [0..]))
==> head (filter (>100) (map (3^) (0:[1..])))   -- evaluate first
         element of the lazy list
```

```
==> head (filter (>100) (1 : map (3^) [1..]))   -- map processes the
        element
==> head (filter (>100) (map (3^) [1..]))       -- filter drops the
        element
==> head (filter (>100) (map (3^) (1:[2..])))   -- evaluate second
        element of the lazy list
==> head (filter (>100) (3 : map (3^) [2..]))   -- map processes the
        element
==> head (filter (>100) (map (3^) [2..]))       -- filter drops the
        element
-- let's take bigger steps now
==> head (filter (>100) (9 : map (3^) [3..]))   -- map processes,
        filter will drop
==> head (filter (>100) (27 : map (3^) [4..]))  -- map processes,
        filter will drop
==> head (filter (>100) (81 : map (3^) [5..]))  -- map processes,
        filter will drop
==> head (filter (>100) (243 : map (3^) [6..])) -- map processes
==> head (243 : filter (>100) (map (3^) [6..])) -- filter lets the
        value through
==> 243                                         -- head returns the
        result
```

10.4 How does Haskell Work?

Laziness will probably feel a bit magical to you right now. You might wonder how it can be implemented. Haskell evaluation is remarkably simple, it's just different than what you might be used to. Let's dig in.

In most other programming languages (like Java, C or Python), evaluation proceeds inside-out. Arguments to functions are evaluated before the function.

Haskell evaluation proceeds outside-in instead of inside-out. The definition of the outermost function in an expression is applied without evaluating any arguments. Here's a concrete example with toy functions `f` and `g`:

```
g :: Int -> Int -> Int
g x y = y+1
f :: Int -> Int -> Int -> Int
f a b c = g (a*1000) c
```

Inside-out (normal) evaluation:

```
f 1 (1234*1234) 2
  -- evaluate arguments to f
  ==> f 1 1522756 2
```

```
  -- evaluate f
  ==> g (1*1000) 2
  -- evaluate arguments to g
  ==> g 1000 2
  -- evaluate g
  ==> 2+1
  ==> 3
```

Haskell outside-in evaluation:

```
f 1 (1234*1234) 2
  -- evaluate f without evaluating arguments
  ==> g (1*1000) 2
  -- evaluate g without evaluating arguments
  ==> 2+1
  ==> 3
```

Note how the unused calculations `1234*1234` and `1*1000` didn't get evaluated. This is why laziness is often helpful.

10.4.1 Pattern Matching Drives Evaluation

Let's look at a more involved example, with pattern matching and more complex data (lists). Pattern matching drives Haskell evaluation in a very concrete way, as we'll see. Here are some functions that we'll use. They're familiar from the Prelude, but I'll give them simple definitions.

```
not True = False
not False = True
map f [] = []
map f (x:xs) = f x : map f xs
length [] = 0
length (x:xs) = 1+length xs
```

Here's the inside-out evaluation of an expression:

```
length (map not (True:False:[]))
  ==> length (not True : not False : [])  -- evaluate call to map
  ==> length (False:True:[])              -- evaluate calls to not
  ==> 2
```

Here's how the evaluation proceeds in Haskell. Note how it's not strictly outside-in, since we sometimes need to evaluate inside arguments to be able to know which

pattern is matched.

```
length (map not (True:False:[]))
  -- We can't evaluate length since we don't know which equation of
        length applies,
  -- so we look at length's argument. We can apply the second
        equation of map, so we do.
  ==> length (not True : map not (False:[]))
  -- Now the argument of length has a (:) we can pattern match on,
        so we apply the
  -- second equation of length
  ==> 1 + length (map not (False:[]))
  -- The outermost function is now +, but it can't do anything
        unless both arguments
  -- are numbers. So we need to evaluate length. In order to pick an
        equation, we need
  -- to evaluate the argument of length again. We apply the second
        equation of map
  ==> 1 + length (not False : map not ([]))
  -- Now we can apply the second equation of length again.
  ==> 1 + (1 + length (map not []))
  -- The outermost + needs a number to be evaluated. The second +
        also needs a number.
  -- We need to evaluate length again, which means we need to pick
        an equation for length,
  -- which means we need to evaluate its argument. This time it is
        the first equation for
  -- map that applies.
  ==> 1 + (1 + length [])
  -- Now we can apply the first equation for length
  ==> 1 + (1 + 0)
  -- The outermost + still can't be evaluated, but the inner one can
  ==> 1 + 1
  -- Finally we evaluate the outer +
  ==> 2
```

Note that we didn't need to evaluate any of the `not` applications.

Let's introduce some terminology. We say that pattern matching *forces* evaluation. When Haskell evaluates something, it evaluates it to something called *weak head normal form (WHNF)*. WHNF basically means *a value that can be pattern matched on*. An expression is in WHNF if it can't be evaluated on *its top level*. This means it either:

- is a constant, for example: `1`
- has a constructor at the top level, for example: `False`, `Just (1+1)`, `0:filter f xs`
- is a function, for example: `(\x -> 1+x)`

The most notable class of expressions that is *not* in WHNF is function applications. If an expression consists of a function (that is not a constructor), applied to some arguments, it is not in WHNF. We must evaluate it in order to get something pattern matchable.

In the previous example we couldn't pick an equation for `length` in `length (map not (False:[]))`. The argument `(map not ...)` is not in WHNF, so it can't be pattern matched on. Thus we need to evaluate it. When we apply the second equation of `map`, we get `length (not False : map not [])`, and now the argument to length is in WHNF since there is a constructor, `(:)`, at the top level. This is a bit more evident if we switch from infix to prefix notation and write the argument to `length` as `(:) (not False) (map not [])`.

In practice, pattern matching is not the only thing that forces evaluation. Primitives like `(+)` also force their arguments.

Instead of forcing, some sources talk about *strictness*, we can say for instance that `(+)` is *strict in both arguments*.

10.4.2 A Word About Sharing

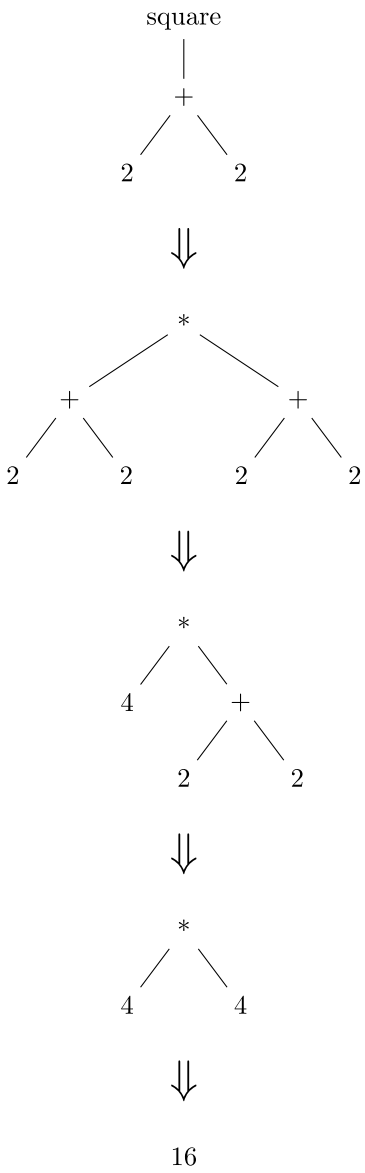
There's one more thing about Haskell evaluation. Any time you give a value a *name*, it gets *shared*. This means that every occurrence of the name points at the same (potentially unevaluated) expression. When the expression gets evaluated, all occurrences of the name see the result.

Let's look at a very simple example.

```
square x = x*x
```

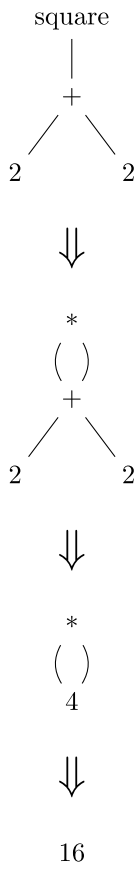
Based on the previous sections, you might imagine evaluation works like the following. The evaluation is first represented textually, and then visually, as an expression tree.

```
square (2+2)
  ==> (2+2) * (2+2)   -- definition of square
  ==>   4   * (2+2)   -- (*) forces left argument
  ==>   4   *   4     -- (*) forces right argument
  ==>      16         -- definition of (*)
```



However, what really happens is that the expression `2+2` named by the variable `x` is only computed once. The result of the evaluation is then shared between the two occurrences of `x` inside `square`. So here' s the correct evaluation, first textually, and then visually. Note how now instead of an expression tree, we have an *expression graph*. This is why Haskell evaluation is sometimes called *graph reduction*.

```
square (2+2)
  ==> (2+2) * (2+2)
  ==>   4   *   4
  ==>       16
```



As another example, consider the function `f` below and its evaluation.

```
f :: Int -> Int
f i = if i>10 then 10 else i
```

```
                  _______shared________
                 |                     |
f (1+1) ==> if (1+1)>10 then 10 else (1+1)
        ==> if 2>10 then 10 else 2
        ==> if False then 10 else 2
        ==> 2
```

Haskell does not compute `1+1` twice because it was named, and the name was used twice. We can contrast this with another function that takes two arguments:

```
g :: Int -> Int
g i j = if i>10 then 10 else j
```

```
                          ______no sharing_____
                         |                     |
g (1+1) (1+1) ==> if (1+1)>10 then 10 else (1+1)
              ==> if 2>10 then 10 else (1+1)
              ==> if False then 10 else (1+1)
              ==> (1+1)
              ==> 2
```

Here we have two different names for equivalent expressions, and Haskell doesn't magically share them. Automatically sharing equivalent expressions is an optimization called *Common Subexpression Elimination (CSE)*. You can learn a bit more CSE and Haskell here.

You can name things via

- Function arguments
- `let ... in ...`
- `where`

Combined with laziness, sharing means that *a name gets evaluated at most once*.

10.4.3 Further Examples

You'll find below a slightly contrived recursive definition of the function `even`. It will illustrate the concepts of forcing and sharing.

```
not :: Bool -> Bool
not True = False
not False = True

(||) :: Bool -> Bool -> Bool
True || _ = True
_    || x = x

even :: Int -> Bool
even x  =  x == 0  ||  not (even (x-1))
```

Firstly, note that `||` forces its left argument, but not its right argument. (In other words, `||` is *strict in its left argument*.) This is because we only need to evaluate the left argument of `||` in order to know which equation applies. This means by extension that `even` forces its first argument:

- `||` forces `x==0`
- `x==0` forces `x`

Now let's evaluate the expression `even 2` to WHNF.

```
even 2
==> 2 == 0  ||  not (even (2-1))                    -- apply
         definition of even
==> False   ||  not (even (2-1))                    -- || forces its
         first argument
==> not (even (2-1))                                -- second
         equation of ||
==> not ((2-1) == 0 || not (even ((2-1)-1)))        -- not forces
         its argument: apply definition of even
==> not (  1   == 0 || not (even (  1  -1)))        -- note sharing!
==> not (  False    || not (even (1-1)))
==> not (not (even (1-1)))
==> not (not ((1-1) == 0 || not (even ((1-1)-1))))
==> not (not (  0   == 0 || not (even (  0  -1))))  -- (sharing)
==> not (not (   True    || not (even (0-1))))
==> not (not True)
==> not False
==> True
```

Note that with this alternate definition `even` would not have worked. Can you tell why?

```
even' x =  not (even' (x-1))  ||  x == 0
```

Now we can really understand what's going on in the infinite list example from earlier. Let's use these definitions:

```
head (x:_) = x
head [] = -1

filter p [] = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs

map f [] = []
map f (x:xs) = f x : map f xs

-- [0..] is syntax sugar for enumFrom 0
enumFrom n = n : enumFrom (n+1)
```

And here we go:

```
    head (filter (>100) (map (3^) [0..]))
=== head (filter (>100) (map (3^) (enumFrom 0)))
-- head forces filter, which forces map, which forces enumFrom. We
        apply the definition of enumFrom.
==> head (filter (>100) (map (3^) (0:[1..])))
-- head forces filter, which forces map. We apply the second
        equation of map.
==> head (filter (>100) ((3^0) : map (3^) [1..]))
-- head forces filter. We apply the second equation of filter
==> head (if ((3^0)>100)
          then (3^0) : filter (>100) (map (3^) [1..])
          else filter (>100) (map (3^) [1..]))
-- head forces if, if forces >, > forces ^. Note sharing!
==> head (if (1>100)
          then 1 : filter (>100) (map (3^) [1..])
          else filter (>100) (map (3^) [1..]))
-- head forces if, if forces >
==> head (if False
          then 1 : filter (>100) (map (3^) [1..])
          else filter (>100) (map (3^) [1..]))
-- apply definition of if
==> head (filter (>100) (map (3^) [1..]))
-- let's take slightly bigger steps now
==> head (filter (>100) (map (3^) (1:[2..])))
==> head (filter (>100) ((3^1) : map (3^) [2..]))
==> head (filter (>100) (3 : map (3^) [2..]))
==> head (filter (>100) (map (3^) [2..]))
-- and even bigger steps now
==> head (filter (>100) (9 : map (3^) [3..]))
==> head (filter (>100) (27 : map (3^) [4..]))
==> head (filter (>100) (81 : map (3^) [5..]))
==> head (filter (>100) (243 : map (3^) [6..]))
==> head (243 : filter (>100) (map (3^) [6..]))
==> 243
```

Whew.

10.5 Working with Infinite Lists

Functions that work with lists often have the best performance when they're written in such a way that they utilize laziness. One way to try to accomplish this is to write list-handling functions that work well with infinite lists.

To write a function that transforms an infinite list, you need to write a function that only looks at a limited prefix of the input list, then outputs a `(:)` constructor, and then recurses. Here's a first example.

```haskell
everySecond :: [a] -> [a]
everySecond [] = []
everySecond (x:y:xs) = x : everySecond xs
```

```haskell
take 10 (everySecond [0..])  ==>  [0,2,4,6,8,10,12,14,16,18]
```

A good heuristic for writing functions that work well with infinite lists is: can the `head` of the result be evaluated cheaply? Here are two examples of functions that don't work with infinite inputs. In the case of `mapTailRecursive`, the problem is that it needs to process the whole input before being in WHNF. In the case of `myDrop`, the problem is that it uses the function `length`, doesn't work for infinite lists since it tries to iterate until the end of the list.

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

mapTailRecursive :: (a -> b) -> [a] -> [b]
mapTailRecursive f xs = go xs []
    where go (x:xs) res = go xs (res++[f x])
          go []     res = res
```

```haskell
head (map inc [0..]) ==> head (inc 0 : map inc [1..]) ==> inc 0 ==>
         1
head (mapTailRecursive inc [0..])
  ==> head (go [0..] [])
  ==> head (go [1..] ([]++[inc 0]))
  ==> head (go [2..] ([]++[inc 0]++[inc 1]))
  ==> head (go [3..] ([]++[inc 0]++[inc 1]++[inc 2]))
  --  never terminates
```

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

myDrop :: Int -> [a] -> [a]
myDrop 0 xs = xs
myDrop n xs = if n > length xs then [] else myDrop (n-1) (tail xs)
```

```
head (drop 2 [0..]) ==> head (drop 1 [1..]) ==> head (drop 0 [2..])
        ==> head [2..] ==> 2
head (myDrop 2 [0..])
  ==> head (if n > length [0..] then [] else myDrop (n-1) (tail
        [0..]))
  ==> head (if n > 1+length [1..] then [] else myDrop (n-1) (tail
        [0..]))
  ==> head (if n > 1+1+length [2..] then [] else myDrop (n-1) (tail
        [0..]))
  ==> head (if n > 1+1+1+length [3..] then [] else myDrop (n-1)
        (tail [0..]))
  --  never terminates
```

Pretty much all the list functions in the standard library are written in this form, for example:

```
head (takeWhile (>=0) [0..]) ==> 0
head (concat (repeat [1,2,3])) ==> 1
head (zip [0..] [2..]) ==> (0,2)
head (filter even [3..]) ==> 4
```

10.6 Interlude: Adding Strictness

Remember `foldr` from part 1? Let's have a look at its cousin `foldl`. Here's the definition of `foldl` for lists (it's actually part of the `Foldable` typeclass and so works for various other types too). While `foldr` processes a list right-to-left, `foldl` processes a list left-to-right. To be a bit more exact, `foldr` *associates to the right* while `foldl` *associates to the left*. Note the difference in the next example:

```
foldr (+) 0 [1,2,3]  ==>  1+(2+(3+0))
foldl (+) 0 [1,2,3]  ==>  ((0+1)+2)+3
```

Here are the definitions of `foldl` and `foldr`:

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

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f y []     = y
```

```
foldr f y (x:xs) = f x (foldr f y xs)
```

As `foldr f y (x:xs) ==> f x (foldr f y xs)`, it enables lazy evaluation to focus on `f` on the second step. Hence, `foldr` works nicely with lazy or short-circuiting operations:

```
    foldr (&&) True [False,False,False]
==> False && (foldr (&&) True [False,False])
==> False
```

```
    head (foldr (++) [] ["Hello","World","lorem","ipsum"])
==> head ("Hello" ++ (foldr (++) [] ["World","lorem","ipsum"]))
==> head ('H':("ello" ++ (foldr (++) [] ["World","lorem","ipsum"])))
==> 'H'
```

However `foldl` needs to process the whole list in order to produce a (WHNF) value. The reason is that `foldl` remains in the leftmost-outermost position for as long as its list argument remains non-empty. This makes `foldl` the priority for lazy evaluation. Only after the list becomes empty does the evaluation proceed into simplifying the folded values.

```
    foldl (&&) True [False,False,False]
==> foldl (&&) (True&&False) [False,False]
==> foldl (&&) ((True&&False)&&False) [False]
==> foldl (&&) (((True&&False)&&False)&&False) []
==> ((True&&False)&&False)&&False
==> (    False    &&False)&&False
==>              False    &&False
==>                      False
```

```
    head (foldl (++) [] ["Hello","World","lorem","ipsum"])
==> head (foldl (++) ([]++"Hello") ["World","lorem","ipsum"])
==> head (foldl (++) (([]++"Hello")++"World") ["lorem","ipsum"])
==> head (foldl (++) ((([]++"Hello")++"World")++"lorem") ["ipsum"])
==> head (foldl (++) (((([]++"Hello")++"World")++"lorem")++"ipsum")
         [])
==> head ((([]++"Hello")++"World")++"lorem")++"ipsum")
-- head forces the last ++, which forces the next-to-last ++, and so
         on
==> head (("Hello"++"World")++"lorem")++"ipsum")
-- same happens again
==> head (('H':("ello"++"World"))++"lorem")++"ipsum")
```

```
-- for clarity, let's drop the "ello"++"World" expression which
        isn't needed
==> head (('H':__)++"lorem")++"ipsum")
-- now the next-to-last ++ can operate
==> head ('H':(__++"lorem"))++"ipsum")
-- let's drop the __++"lorem" expression
==> head ('H':__)++"ipsum")
-- now the last ++ can operate
==> head ('H':(__++"ipsum"))
==> 'H'
```

So why use `foldl` at all? Let's return to our first fold example again. Now, since + is a strict operation, both types of fold need to build up an expression with lots of +s. The Haskell implementation needs to track this expression in memory, which is why a problem like this is called a *space leak*.

```
    foldr (+) 0 [1,2,3]
==> 1 + foldr (+) 0 [2,3]
==> 1 + (2 + foldr (+) 0 [3])
==> 1 + (2 + (3 + foldr (+) 0 []))
==> 1 + (2 + (3 + 0))
==> 1 + (2 + 3)
==> 1 + 5
==> 6
```

```
    foldl (+) 0 [1,2,3]
==> foldl (+) (0+1) [2,3]
==> foldl (+) ((0+1)+2) [3]
==> foldl (+) (((0+1)+2)+3) []
==> ((0+1)+2)+3
==> (  1  +2)+3
==>       3  +3
==>          6
```

Now let's instead look at what happens when we use `foldl'`, a version of `foldl` that forces its second argument!

```
    foldl' (+) 0 [1,2,3]
==> foldl' (+) (0+1) [2,3]
-- force second argument
==> foldl' (+) 1 [2,3]
==> foldl' (+) (1+2) [3]
-- force second argument
==> foldl' (+) 3 [3]
==> foldl' (+) (3+3) []
```

```
-- force second argument
==> foldl' (+) 6 []
==> 6
```

Now the work is performed incrementally while scanning the list. No space leak! Sometimes too much laziness can cause space leaks, and a bit of strictness can fix them.

You can find `foldl'` in the `Data.List` module, and it works just like this. But how could one implement `foldl'`? We certainly know by now how to do it for a specific type, say `Int`. We just add a pattern match on the second argument that doesn't change the semantics of the function.

```
foldl'Int :: (Int -> Int -> Int) -> Int -> [Int] -> Int
foldl'Int f z [] = z
foldl'Int f 0 (x:xs) = foldl'Int f (f 0 x) xs
foldl'Int f z (x:xs) = foldl'Int f (f z x) xs
```

```
    foldl'Int (+) 0 [1,2,3]
==> foldl'Int (+) (0+1) [2,3]
-- to be able to pick between the second and third equations, (0+1)
         is forced
==> foldl'Int (+) 1 [2,3]
-- the third equation applies
==> foldl'Int (+) (1+2) [3]
-- again, we need to pick between the second and third equations
==> foldl'Int (+) 3 [3]
==> foldl'Int (+) (3+3) []
==> 3+3
==> 6
```

To write a generic implementation of `foldl'` we need to introduce a new built-in function, `seq`. The call `seq a b` evaluates to `b` but forces `a` into WHNF. Here are some examples of using `seq` in GHCi. To demonstrate what gets evaluated, we use the special value `undefined`, which causes an error if something tries to evaluate it into WHNF.

```
Prelude> seq (not True) 3
3
Prelude> seq undefined 3
*** Exception: Prelude.undefined
Prelude> (seq (not True) 3) + 7
10
Prelude> (seq undefined 3) + 7
```

```
*** Exception: Prelude.undefined
Prelude> let f x = f x in seq (f 3) 3
-- ...infinite recursion
```

As an example of using `seq` in a function, here's a version of `head` that doesn't work for infinite lists (since it evaluates the last element of the list):

```
strictHead :: [a] -> a
strictHead xs = seq (last xs) (head xs)
```

Let's play around with it in GHCi:

```
Prelude> head [1,2,3]
1
Prelude> strictHead [1,2,3]
1
Prelude> head (1:2:undefined)
1
Prelude> strictHead (1:2:undefined)
*** Exception: Prelude.undefined
Prelude> head [1..]
1
Prelude> strictHead [1..]
-- ...infinite recursion
```

Finally, here's a definition for `foldl'`. Note how we need to introduce sharing of a new variable, `z'`, to be able to make `seq` evaluate the new value and then use it in a recursive call. The new definition is also used in a more detailed evaluation of `foldl' (+) 0 [1,2,3]` below.

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f z [] = z
foldl' f z (x:xs) = let z' = f z x
                    in seq z' (foldl' f z' xs)
```

```
    foldl' (+) 0 [1,2,3]
==> seq (0+1) (foldl' (+) (0+1) [2,3])  -- seq forces first argument
          |                  |
          +-----sharing------'
          |                  |
==> seq   1   (foldl' (+)   1   [2,3])  -- first argument to seq in
        WHNF, seq disappears
```

```
==> foldl' (+) 1 [2,3]
==> seq (1+2) (foldl' (+) (1+2) [3])
==> seq   3   (foldl' (+)   3   [3])
==> foldl' (+)   3   [3]
==> seq (3+3) (foldl' (+) (3+3) [])
==> seq   6   (foldl' (+)   6   [])
==> foldl' (+)   6   []
==> 6
```

We won't dive deeper into this subject on this course, but it's important that you're aware that `seq` exists. You can find more about `seq` on the Haskell Wiki and learn more about when it is necessary to add strictness in Real World Haskell. Often it's nicer to use *bang patterns* instead of `seq`, as discussed by FPComplete and Real World Haskell.

10.7 Newtype Declarations

Recall lecture 7. Sometimes we need boxed types. There's a special keyword `newtype` that can be used instead of `data` when a boxed type is needed. `newtype` expects exactly one constructor, with exactly one field. For instance,

```
newtype Money = Cents Int
```

However, the following won't work, you need `data`:

```
-- the compiler won't accept these!
newtype Currency = Dollars Int | Euros Int
newtype Money = Money Int Int
```

So what's the difference? In terms of writing code, nothing. You work with a `newtype` exactly as you would with a `data`. However, the memory layout is different. Using `data` introduces an indirection layer (the constructor), but using `newtype` doesn't. The indirection for `data` is necessary to support multiple constructors and multiple fields. An illustration:

```
code:                          memory:

data Money = Cents Int         x --> Cents --> 100
x = Cents 100
```

```
newtype Money = Cents Int              x --> 100
x = Cents 100
```

This difference has many repercussions. First of all, `newtype` is more efficient: the type can be said to "disappear" when compiling. The type is still checked though, so you get type safety without any performance impact. Secondly, newtypes are *strict*. Concretely, this means that `Money x` is in weak head normal form only if `x` is in WHNF. This can be witnessed in GHCi:

```
-- if we use data, Cents undefined is in WHNF
Prelude> data Money = Cents Int
Prelude> seq (Cents undefined) True
True
-- if we use newtype, Cents undefined isn't in WHNF, and trying
-- to make it so trips up in undefined
Prelude> newtype Money = Cents Int
Prelude> seq (Cents undefined) True
*** Exception: Prelude.undefined
```

So when should you use `newtype`? In general it's best to use `newtype` whenever you have a single-field single-constructor datatype. However nothing will go catastrophically wrong if you always use `data`. The `newtype` pattern is also often used when you need to define a different type class instance for a type. Here's an example that defines a number type with an inverted ordering

```
newtype Inverted = Inverted Int
  deriving (Show, Eq)

instance Ord Inverted where
  compare (Inverted i) (Inverted j) = compare j i
```

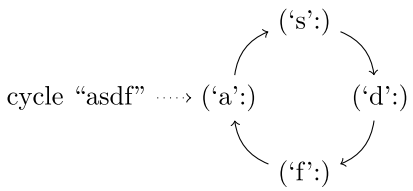
```
Prelude Data.List> sort [1,2,3]
[1,2,3]
Prelude Data.List> sort [Inverted 1,Inverted 2,Inverted 3]
[Inverted 3,Inverted 2,Inverted 1]
```

10.8 Something Fun: Tying the Knot

Now that we know about sharing and path copying, we can make our own *cyclic datastructures*. Remember the `cycle` examples from the list lecture?

```
Prelude> take 21 $ cycle "asdf"
"asdfasdfasdfasdfasdfa"
```

This is what it looks like in memory:



Earlier it was said that Haskell data forms directed graphs in memory. This is an example of a directed graph with a cycle.

How can we define structures like this? We just give a value a name, and refer to that name in the value itself. That is, the value is *recursive* or *self-referential*. This trick is known as *tying the knot*. A simple example:

```
  code                      memory

let xs = 1:2:xs      xs -> (1:) -> (2:) -+
 in xs                       ^            |
                             +------------+
```

Note how we use the name `xs` inside the definition of `xs`. When we make a recursive definition like this, sharing causes it to turn into a cyclic structure in memory.

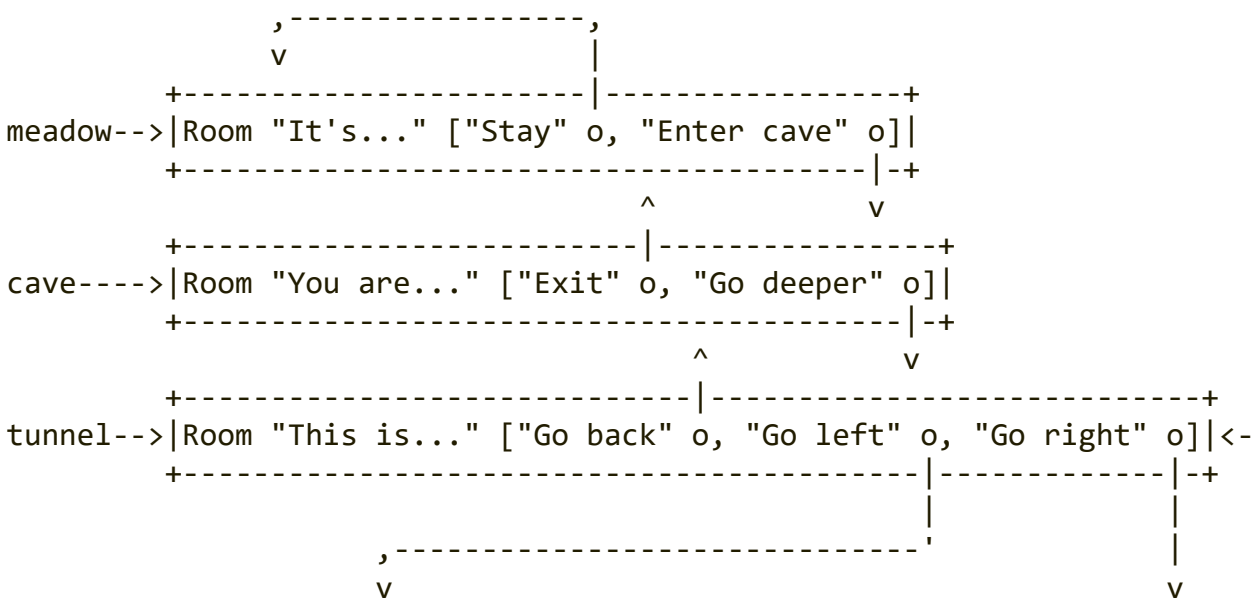
A more fun example: a simple adventure game where the world is a self-referential structure. Note how the cyclic structure is built with local definitions that refer to each other.

```
data Room = Room String [(String,Room)]

describe :: Room -> String
describe (Room s _) = s

move :: Room -> String -> Maybe Room
move (Room _ directions) direction = lookup direction directions

world :: Room
world = meadow
  where
    meadow = Room "It's a flowery meadow next to a cliff."
         [("Stay",meadow),("Enter cave",cave)]
```

```
    cave = Room "You are in a cave" [("Exit",meadow),("Go
         deeper",tunnel)]
    tunnel = Room "This is a very dark tunnel. It seems you can
         either go left or right."
                  [("Go back",cave),("Go left",pit),("Go
         right",treasure)]
    pit = Room "You fall into a pit. There is no way out." []
    treasure = Room "A green light from a terminal fills the room.
         The terminal says <<loop>>."
                    [("Go back",tunnel)]

play :: Room -> [String] -> [String]
play room [] = [describe room]
play room (d:ds) = case move room d of Nothing -> [describe room]
                                       Just r -> describe room :
         play r ds
```

```
Prelude> play world ["Stay","Enter cave","Go deeper","Go back","Go
         deeper","Go right"]
["It's a flowery meadow next to a cliff.",
 "It's a flowery meadow next to a cliff.",
 "You are in a cave",
 "This is a very dark tunnel. It seems you can either go left or
         right.",
 "You are in a cave",
 "This is a very dark tunnel. It seems you can either go left or
         right.",
 "A green light from a computer terminal floods the room. The
         terminal says <<loop>>."]
```

Here's what the `world` of the game looks like in memory:



```
          +---------------------+                +----------------------
pit----->|Room "You fall..." []|    treasure--->|Room "A green..." ["Go
          +---------------------+                +----------------------
```

We've now seen three types of recursion. Recursive functions call themselves. Recursive types allow us to express arbitarily large structures. Recursive values are one way to implement infinite structures.

10.9 Something Fun: Debug.Trace

Even though Haskell is a pure programming language, we can sometimes gain insights by sprinkling in a bit of impurity.

We can use the function `trace :: String -> a -> a` from the module `Debug.Trace` to peek into Haskell evaluation. The expression `trace "message" x` is the same as `x`, but prints `message` when it is evaluated (forced). We can use `trace` to witness the laziness of the `||` operator:

```
Prelude> import Debug.Trace
Prelude Debug.Trace> trace "a" True
a
True
Prelude Debug.Trace> trace "a" False || trace "b" True
a
b
True
Prelude Debug.Trace> trace "a" True || trace "b" True
a
True
```

We can also have a look at when list elements are evaluated. Note how `length` doesn't need to evaluate the elements of the list, and `sum` needs to evaluate all of them. (To be precise, `head xs` doesn't actually evaluate the first element of `xs`, but returns it to GHCi, which evaluates it in order to show it.)

```
Prelude Debug.Trace> head [trace "first" 1, trace "second" 2, trace
         "third" 3]
first
1
Prelude Debug.Trace> last [trace "first" 1, trace "second" 2, trace
         "third" 3]
third
3
Prelude Debug.Trace> length [trace "first" 1, trace "second" 2,
         trace "third" 3]
```

```
3
Prelude Debug.Trace> sum [trace "first" 1, trace "second" 2, trace
         "third" 3]
third
second
first
6
```

`Debug.Trace` also offers useful variants of `trace`. A notable one is `traceShowId x` which prints `show x` and evaluates to `x`. Let's verify the evaluation of our previous head-filter-map example using `traceShowId`. Note how even though we map `traceShowId` over the infinite list `[0..]`, only 6 values are actually evaluated. The last 243 is the returned value, not a trace print.

```
Prelude Debug.Trace> head (filter (>100) (map (\x -> traceShowId
         (3^x)) [0..]))
1
3
9
27
81
243
243
```

`Debug.Trace` is especially useful when you have an infinite recursion bug. Here's an example:

```
-- computes sums like 7+5+3+1
sumEverySecond :: Int -> Int
sumEverySecond 0 = 0
sumEverySecond n = n + sumEverySecond (n-2)
```

```
sumEverySecond 6 ==> 12
sumEverySecond 7 ==> doesn't terminate
```

We can debug this by adding a `trace` to wrap the whole recursive case.

```
sumEverySecond :: Int -> Int
sumEverySecond 0 = 0
sumEverySecond n = trace ("sumEverySecond "++show n) (n +
         sumEverySecond (n-2))
```

```
Prelude Debug.Trace> sumEverySecond 6
sumEverySecond 6
sumEverySecond 4
sumEverySecond 2
12
Prelude Debug.Trace> sumEverySecond 7
sumEverySecond 7
sumEverySecond 5
sumEverySecond 3
sumEverySecond 1
sumEverySecond -1
sumEverySecond -3
sumEverySecond -5
-- and so on
```

A ha! The problem is that our recursion base case of `sumEverySecond 0` is not enough to stop the recursion.

Finally, a word of caution. Using `trace`, and especially `traceShowId`, can cause things that would not otherwise get evaluated to get evaluated. For example:

```
Prelude Debug.Trace> traceHead xs = head (traceShowId xs)
Prelude Debug.Trace> traceHead [0..]
-- never terminates since it's trying to show an infinite list
```

So feel free to use `Debug.Trace` when working on the exercises, but try to leave `trace` calls out of your final answers. Some exercise sets check your imports and disallow `Debug.Trace`.

We'll see a more principled way of dealing with side effects in the next lecture!

10.10 Quiz

Which of these statements is true?

a. `reverse . reverse . reverse === reverse`
b. `reverse . reverse === reverse`
c. `reverse . id === id`

Which of these is an infinite list that starts with `[0,1,2,1,2,1,2...]`?

a. `cycle [0,1,2]`
b. `0:repeat [1,2]`
c. `0:cycle [1,2]`

d. `0:[1,2..]`

What's the next step when evaluating this expression?

```
head (map not (True:False:[]))
```

a. `head (False : True : [])`
b. `head (not True)`
c. `head (False : map not (False:[]))`
d. `head (not True : map not (False:[]))`

Which of these values is *not* in weak head normal form?

a. `map`
b. `f 1 : map f (2 : [])`
c. `Just (not False)`
d. `(\x -> x) True`

Which of these statements about the following function is true?

```
f 0 x = 1+x
f _ x = 2+x
```

a. `f` is strict in its left argument
b. `f` is strict in its right argument
c. `f` forces both of its arguments
d. None of the above

Does this function work with infinite lists as input? Why?

```
f [] = []
f (x:xs) = x : map not xs
```

a. No, because it includes a `[]` case, which is never reached.
b. No, because it uses `map`, which evaluates the whole list.
c. Yes, because it only looks at the first element of the list before producing a WHNF value.
d. Yes, because it calls `map`, which works with infinite lists.

What about this one?

```
f xs = map (+(sum xs)) xs
```

a. No, because it uses `map`, which evaluates the whole list.
b. No, because computing the `head` of the result needs the whole input list.
c. Yes, because it doesn't includes a `[]` case
d. Yes, because it calls `map`, which works with infinite lists.

10.11 Exercises

- Set10a
- Set10b

11 Lecture 11: `RealWorld -> (a,RealWorld)`

11.1 Contents

- IO

11.2 You've Been Fooled!

Forget what we talked about functional programming and purity. Actually, Haskell is the *world's best imperative programming language*! Let's start:

```
questionnaire = do
  putStrLn "Write something!"
  s <- getLine
  putStrLn ("You wrote: "++s)
```

```
Prelude> questionnaire
Write something!
Haskell!
You wrote: Haskell!
```

Reading input and writing output was easy enough. We can also read stuff over the network. Here's a complete Haskell program that fetches a some words from a URL using HTTP and prints them.

```
import Network.HTTP
import Control.Monad

main = do
  rsp <- simpleHTTP (getRequest
         "http://httpbin.org/base64/aGFza2VsbCBmb3IgZXZlcgo=")
  body <- getResponseBody rsp
  forM_ (words body) $ \w -> do
     putStr "word: "
     putStrLn w
```

You can find this program in the course repository as `exercises/Examples/FetchWords.hs`, and you can run it like this:

```
$ cd exercises/Examples
$ stack runhaskell FetchWords.hs
word: haskell
word: for
word: ever
```

What's going on here? Let's look at the types:

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude> :t getLine
getLine :: IO String
```

A value of type `IO a` is an *operation* that *produces* a value of type `a`. So `getLine` is an IO operation that produces a string. The `()` type is the so called *unit type*, its only value is `()`. It's mostly used when an IO operation doesn't return anything (but rather just has side effects).

A comparison with Java (method) types might help:

Haskell type	**Java type**
`doIt :: IO ()`	`void doIt()`
`getSomething :: IO Int`	`int getSomething()`
`force :: a -> b -> IO ()`	`void force(a arg0, b arg1)`
`mogrify :: c -> IO d`	`d mogrify(c arg)`

IO operations can be combined into bigger operations using *do-notation*.

```
do operation
   operation arg
   variable <- operationThatReturnsStuff
   let var2 = expression
   operationThatProducesTheResult var2
```

11.2.1 Examples

You can find useful IO operations in the standard library modules Prelude and System.IO

Here’ s an IO operation that asks the user for a string, and prints out the length of the string.

```
query :: IO ()
query = do
  putStrLn "Write something!"                    -- run an
         operation, ignore produced value
  s <- getLine                                   -- run an
         operation, capture produced value
  let n = length s                               -- run a pure
         function
  putStrLn ("You wrote "++show n++" characters") -- run an
         operation, passing on the produced value
```

```
Prelude> query
Write something!
lorem ipsum
You wrote 11 characters
```

The value produced by the last line of a `do` block is the value produced by the whole block. Note how `askForALine` has the same type as `getLine`, `IO String`:

```
askForALine :: IO String
askForALine = do
  putStrLn "Please give me a line"
  getLine
```

In addition to IO operations like `query` you can also run IO operations that produce values, like `askForALine`, in GHCi. You can use `<-` to capture the result of the operation into a variable if you want.

```
Prelude> askForALine
Please give me a line
this is a line
"this is a line"
Prelude> line <- askForALine
Please give me a line
this is a line
Prelude> :t line
line :: String
Prelude> line
"this is a line"
```

If you need to give your operation parameters, you can just make *a function that returns an operation*. Note how `ask` has a function type with a `->`, just like a normal function. We also use normal function definition syntax to give the parameter the name `question`.

```
ask :: String -> IO String
ask question = do
  putStrLn question
  getLine
```

```
Prelude> ask "What is love?"
What is love?
Baby don't hurt me!
"Baby don't hurt me!"
Prelude> response <- ask "Who are you?"
Who are you?
The programmer.
Prelude> response
"The programmer."
Prelude> :t response
response :: String
Prelude> :t ask
ask :: String -> IO String
Prelude> :t ask "Who are you?"
ask "Who are you?" :: IO String
```

11.3 The Subtle `return`

The Haskell function `return` is named a bit misleadingly. In other languages `return` is a built-in keyword, but in Haskell it's just a function. The `return :: a -> IO a` function takes a value and turns it into an *operation, that produces the value*.

```haskell
produceThree :: IO Int
produceThree = return 3

printThree :: IO ()
printThree = do
  three <- produceThree
  putStrLn (show three)
```

That doesn't sound very useful does it? Combined with do-notation it is. Here we return a boolean according to whether the user answered `Y` or `N`:

```haskell
yesNoQuestion :: String -> IO Bool
yesNoQuestion question = do
  putStrLn question
  s <- getLine
  return (s == "Y")
```

```
Prelude> yesNoQuestion "Fire the missiles?"
Fire the missiles?
Y
True
Prelude> answer <- yesNoQuestion "Are you sure?"
Are you sure?
N
Prelude> :t answer
answer :: Bool
Prelude> answer
False
```

Note! This means that return *does not stop execution* of an operation (unlike return in Java or C). Remember that in do-blocks, the last line decides which value to produce. This means that this operation produces `2`:

```haskell
produceTwo :: IO Int
produceTwo = do return 1
                return 2
```

```
Prelude> produceTwo
2
```

Let's look at this another way. The `do` notation allows us to cause a sequence of side-effects, and finally to produce a value.

```
produceThree = do putStrLn "1"   -- side effect, produces (), which is
                  return    2    -- no side effect, produces 2, which i
                  getLine        -- side effect, produces a String, whi
                  return    3    -- no side effect, produces 3, which i
```

```
Prelude> final <- produceThree
1
this line is ignored
Prelude> final
3
```

Also note that these are the same operation:

```
do ...
   x <- op
   return x
```

```
do ...
   op
```

Since `return` is a function, you should remember to parenthesize any complex expressions:

```
return (f x : xs)
-- alternatively:
return $ f x : xs
```

11.4 do and Types

Let's look at the typing of do-notation in more detail. A do-block builds a value of type `IO <something>`. For example in

```
foo = do
  ...
  lastOp
```

The `lastOp` must be of type `IO X` (for some `X`). The type of `foo` will also be `IO X`. Let's look at an example with parameters next:

```
bar x y = do
  ...
  lastOp arg
```

The `lastOp` must be of type `Y -> IO X` (so that `lastOp arg` has type `IO X`). The type of `bar` will be `A -> B -> IO X` (and inside `bar` we'll have `x :: A` and `y :: B`).

If we use `return`:

```
quux x = do
  ...
  return value
```

The function `quux` will have type `A -> IO B`, where `x :: A` and `value :: B`.

Let's look at the typing of `<-` next. If `op :: IO X` and you have `var <- op`, `var` will have type `X`. We've seen this in many GHCi examples.

The last line of a `do` cannot be `foo <- bar`. It can't be `let foo = bar` either. The last line determines what the whole operation produces, so it must be an operation (for example, `return something`).

Here's a worked example:

```
alwaysFine :: IO Bool
alwaysFine = do
  putStrLn "What?" -- :: IO ()
  return 2         -- :: IO Int, produced value is discarded
  s <- getLine     -- getLine :: IO String, thus s :: String
  putStrLn s       -- putStrLn :: String -> IO (), thus putStrLn s
         :: IO ()
  let b = True     -- b :: Bool
  return b         -- :: IO Bool
                   -- Thus, alwaysFine :: IO Bool
```

The typing rules guarantee that you can not "escape" `IO`. Even though `<-` gives you an `X` from an `IO X`, you can only use `<-` inside `do`. However a `do` always means a value of type `IO Y`. In other words: you can temporarily open the `IO` box, but you must return into it. *"What happens in IO, stays in IO."*

We'll talk more about what this means later. For now, it's enough to know that if you have a function with a non-IO type, like for example `myFunction :: Int -> [String] -> String`, the function can not have IO happening inside it. It is a pure function.

11.5 Control Structures

For the following examples, we'll need two new operations.

```
print :: Show a => a -> IO ()    -- print a value using the show
         function
readLn :: Read a => IO a        -- get a line and convert it to a
         value using the read function
```

The usual tools of recursion, guards and if-then-else also work in the `IO` world. Here's an IO operation that's defined using a guard:

```
printDescription :: Int -> IO ()
printDescription n
  | even n    = putStrLn "even"
  | n==3      = putStrLn "three"
  | otherwise = print n
```

```
Prelude> printDescription 2
even
Prelude> printDescription 3
three
Prelude> printDescription 5
5
```

Here's an operation that prints all numbers in a list using recursion and pattern matching:

```
printList :: [Int] -> IO ()
printList [] = return () -- do nothing
```

```
printList (x:xs) = do print x
                      printList xs -- recursion
```

```
Prelude> printList [1,2,3]
1
2
3
```

Here are two slightly more complicated examples of recursive IO operations. They use the value produced by the recursive call. The operation `readAndSum n` reads `n` numbers from the user and prints their sum. The operation `ask questions` shows each string in `questions` to the user, reads a response, and returns a list of all the responses.

```
readAndSum :: Int -> IO Int
readAndSum 0 = return 0
readAndSum n = do
  i <- readLn             -- read one number
  s <- readAndSum (n-1)   -- recursion: read and sum rest of numbers
  return (i+s)            -- produce result
```

```
Prelude> s <- readAndSum 3
2
4
5
Prelude> s
11
```

```
ask :: [String] -> IO [String]
ask [] = return []
ask (question:questions) = do
  putStr question
  putStrLn "?"
  answer <- getLine          -- get one answer
  answers <- ask questions   -- recursion: get rest of answers
  return (answer:answers)    -- produce result
```

```
Prelude> replies <- ask ["What is your name","How old are you"]
What is your name?
Yog-Sothoth
```

```
How old are you?
The question is meaningless
Prelude> replies
["Yog-Sothoth","The question is meaningless"]
```

Additionally, we have some `IO`-specific control structures, or rather, functions. These come from the module `Control.Monad`.

```
-- when b op performs op if b is true
when :: Bool -> IO () -> IO ()
-- unless b op performs op if b is false
unless :: Bool -> IO () -> IO ()
-- do something many times, collect results
replicateM :: Int -> IO a -> IO [a]
-- do something many times, throw away the results
replicateM_ :: Int -> IO a -> IO ()
-- do something for every list element
mapM :: (a -> IO b) -> [a] -> IO [b]
-- do something for every list element, throw away the results
mapM_ :: (a -> IO b) -> [a] -> IO ()
-- the same, but arguments flipped
forM  :: [a] -> (a -> IO b) -> IO [b]
forM_ :: [a] -> (a -> IO b) -> IO ()
```

Using these, we can rewrite our earlier examples:

```
printList :: [Int] -> IO ()
printList xs = mapM_ print xs
```

```
readAndSum n = do
  numbers <- replicateM n readLn
  return (sum numbers)
```

```
ask :: [String] -> IO [String]
ask questions = do
  forM questions askOne

askOne :: String -> IO String
askOne question = do
  putStr question
```

```
  putStrLn "?"
  getLine
```

11.6 A Word About `do` and Indentation

It's easy to run into weird indentation problems when using do-notation. Here are some rules of thumb to help you get it right.

The most important rule of do and indentation is *all operations in a do-block must start in the same column*.

Some examples of this rule:

```
-- This is not OK, putStrLn is way too left
foo = do y <- getLine
   putStrLn y

-- This is not OK either
foo = do y <- getLine
           putStrLn y

-- This is OK
foo = do y <- getLine
         putStrLn y

-- This is also OK: putting a line break after do
foo = do
  y <- getLine
  putStrLn y
```

A related rule is *when an operation goes over multiple lines, indent the follow-up lines*. If you don't indent, it'll look like a new operation!

```
-- This is not OK, the string starts a new operation
quux = do putStrLn
          "this long string"
          print 1

-- This is OK
quux = do putStrLn
            "this long string"
          print 1
```

Here's one more example, with nested do-blocks, and two different valid indentations.

```haskell
-- This is OK
foo x = do quux
           y <- blorg
           when y (do thing
                      otherThing)
           return 3

-- This is also OK: starting putting a line break after do, using $
foo x = do
  quux
  y <- blorg
  when y $ do
    thing
    otherThing
  return 3
```

11.7 Let's Write a Program

After all these short one-off examples, let's turn to something a bit longer. Let's write a program to fetch all type annotations from all `.hs` files. We use IO operations like `readFile` and `listDirectory` to read and find files, but also pure code like `map` and `filter` to do the actual processing. First off, here's a recap of the library operations we're using:

```haskell
-- split string into lines
lines :: String -> [String]
-- `isSuffixOf suf list` is true if list ends in suf
Data.List.isSuffixOf :: Eq a => [a] -> [a] -> Bool
-- `isInfixOf inf list` is true if inf occurs inside list
Data.List.isInfixOf :: Eq a => [a] -> [a] -> Bool
-- FilePath is just an alias for String
type FilePath = String
-- get entire contents of file
readFile :: FilePath -> IO String
-- list files in directory
System.Directory.listDirectory :: FilePath -> IO [FilePath]
-- is the given file a directory?
System.Directory.doesDirectoryExist :: FilePath -> IO Bool
```

And here's the program itself. You can also find it in the course repository as `exercises/Examples/ReadTypes.hs`.

```haskell
module Examples.ReadTypes where
```

```
import Control.Monad (forM)
import Data.List (isInfixOf, isSuffixOf)
import System.Directory (listDirectory, doesDirectoryExist)

-- a line is a type signature if it contains :: but does not contain
         =
isTypeSignature :: String -> Bool
isTypeSignature s = not (isInfixOf "=" s) && isInfixOf "::" s

-- return list of types for a .hs file
readTypesFile :: FilePath -> IO [String]
readTypesFile file
  | isSuffixOf ".hs" file = do content <- readFile file
                               let ls = lines content
                               return (filter isTypeSignature ls)
  | otherwise             = return []

-- list children of directory, prepend directory name
qualifiedChildren :: String -> IO [String]
qualifiedChildren path = do childs <- listDirectory path
                            return (map (\name -> path++"/"++name)
         childs)

-- get type signatures for all entries in given directory
-- note mutual recursion with readTypes
readTypesDir :: String -> IO [String]
readTypesDir path = do childs <- qualifiedChildren path
                       typess <- forM childs readTypes
                       return (concat typess)

-- recursively read types contained in a file or directory
-- note mutual recursion with readTypesDir
readTypes :: String -> IO [String]
readTypes path = do isDir <- doesDirectoryExist path
                    if isDir then readTypesDir path else
         readTypesFile path

-- main is the IO action that gets run when you run the program
main :: IO ()
main = do ts <- readTypes "."
          mapM_ putStrLn ts
```

We can run this program by going to the directory `exercises/Examples` and running:

```
$ stack runhaskell ReadTypes.hs
deposit :: String -> Int -> Bank -> Bank
withdraw :: String -> Int -> Bank -> (Int,Bank)
```

```
runBankOp :: BankOp a -> Bank -> (a,Bank)
... and so on
```

The exact output will vary according to the contents of the directory, of course.

11.8 What Does It All Mean?

Let's return to the functional world. How can we reconcile IO operations with Haskell being a *pure* and *lazy* language? Something like `putStrLn :: String -> IO ()` is a *pure* function that returns an operation. How is it pure? `putStrLn x` is the same when `x` is the same. In other words: an operation is a *pure description* of a chain of side effects. Only *executing* the operation causes those side effects. When a Haskell program is run, only one operation is executed - it's called `main :: IO ()`. Other operations can be run only by linking them up to `main`.

When in GHCi, if an expression you type in evaluates to an operation, GHCi runs that operation for you. Here's a demonstration of the purity of `print`:

```
Prelude> x = print 1   -- creates operation, doesn't run it
Prelude> x             -- runs the operation
1
Prelude> x             -- runs it again!
1
```

Operations are values just like numbers, lists and functions. We can write code that operates on operations. This function takes two operations, `a` and `b`, and returns an operation that asks the user which one he'd like to run.

```
choice :: IO x -> IO x -> IO x
choice a b =
  do putStr "a or b? "
     x <- getLine
     case x of "a" -> a
               "b" -> b
               _ -> do putStrLn "Wrong!"
                       choice a b
```

```
Prelude> choice (putStrLn "A!!!!") (putStrLn "B!!!!")
a or b? z
Wrong!
```

```
a or b? a
A!!!!
```

Using operations specified as parameters lets us write functions like `mapM_`, which we met earlier. The implementation is a recursive IO operation that takes another IO operation as a parameter. Conceptually complicated, but simple when you read the code:

```
mapM_ :: (a -> IO b) -> [a] -> IO ()
mapM_ op     [] = return ()       -- do nothing for an empty list
mapM_ op (x:xs) = do op x         -- run operation on first element
                     mapM_ op xs  -- run operation on rest of list,
        recursively
```

```
Prelude> mapM_ print [1,2,3]
1
2
3
```

11.9 One More Thing: IORef

So far the only side-effects we've been able to produce in IO have been terminal (`getLine`, `print`) and file (`readFile`, `listDirectory`) IO. Imperative programs written in Java, Python or C have other types of side effects too that we can't express in pure Haskell. One of these is *mutable (i.e. changeable) state*. A pure function can not read mutable state, because otherwise two invocations of the same function might not return the same value.

The Haskell type `IORef a` from the module `Data.IORef` is a mutable reference to a value of type `a`

```
newIORef :: a -> IO (IORef a)              -- create a new IORef
        containing a value
readIORef :: IORef a -> IO a               -- produce value
        contained in IORef
writeIORef :: IORef a -> a -> IO ()        -- set value in IORef
modifyIORef :: IORef a -> (a -> a) -> IO ()  -- modify value
        contained in IORef with a pure function
```

Here are some examples of using an IORef in GHCi:

```
Prelude> :m +Data.IORef
Prelude Data.IORef> myRef <- newIORef "banana"
Prelude Data.IORef> readIORef myRef
"banana"
Prelude Data.IORef> writeIORef myRef "apple"
Prelude Data.IORef> readIORef myRef
"apple"
Prelude Data.IORef> modifyIORef myRef reverse
Prelude Data.IORef> readIORef myRef
"elppa"
```

Here's an example of using an `IORef` to sum the values in a list. Note the similarity with an imperative loop.

```
sumList :: [Int] -> IO Int
sumList xs = do r <- newIORef 0                          -- initialize
         r to 0
                forM_ xs (\x -> modifyIORef r (x+))      -- for every
         xs, add it to r
                readIORef r                              -- get last
         value of r
```

Using `IORef` isn't necessary most of the time. Haskell style prefers recursion, arguments and return values. However real world programs might need one or two IORefs occasionally.

11.10 Summary of IO

A value of type `IO X` is an *IO operation* that *produces* a value of type X *when run*. Operations are pure values. Only *running* the operation causes the side effects.

IO operations can combined together using `do`-notation:

```
op :: X -> IO Y
op arg = do operation                -- run operation
            operation2 arg           -- run operation with argument
            result <- operation3 arg -- run operation with
         argument, store result
            let something = f result -- run a pure function f,
         store result
            finalOperation           -- last operation produces the
         the return value
```

The `return x` operation is an operation that always produces value `x`. When `x :: a`, `return x :: IO a`.

Useful IO operations:

```
-- printing & reading
putStr :: String -> IO ()
putStrLn :: String -> IO ()
print :: Show a => a -> IO ()
getLine :: IO String
readLn :: Read a => IO a

-- control structures from Control.Monad
when :: Bool -> IO () -> IO ()   -- when b op performs op if b is
         true
unless :: Bool -> IO () -> IO () -- unless b op performs op if b is
         false
replicateM :: Int -> IO a -> IO [a]   -- do something many times,
         collect results
replicateM_ :: Int -> IO a -> IO ()   -- do something many times,
         throw away the results
mapM ::  (a -> IO b) -> [a] -> IO [b] -- do something for every list
         element
mapM_ :: (a -> IO b) -> [a] -> IO ()  -- do something for every list
         element, throw away the results
forM ::  [a] -> (a -> IO b) -> IO [b] -- the same, but arguments
         flipped
forM_ :: [a] -> (a -> IO b) -> IO ()

-- files
readFile :: FilePath -> IO String
```

11.11 Quiz

What is the type of this IO operation?

```
foo x = do putStrLn x
           y <- getLine
           return (length y)
```

a. `String -> IO String`
b. `IO Int`
c. `String -> IO Int`
d. `IO String -> IO Int`

Which of these lines could be used in place of `????`

```
quux :: String -> IO [String]
quux q = do y <- getLine
            z <- getLine
            putStrLn (y++z)
            ????
```

a. `q <- getLine`
b. `return (y++z)`
c. `return [q]`
d. `ans <- return [y,z]`

What values does `blorg [1,2,3]` print? That is, what values `x` does it call `print x` for. The value produced by `blorg` doesn't count.

```
blorg [] = return 0
blorg (x:xs) = do m <- blorg xs
                  print x
                  return (m+x)
```

a. It prints 1, 2, 3
b. It prints 1, 2, 3, 6
c. It prints 3, 2, 1
d. It prints 3, 2, 1, 6

Which of these can a function of type `Int -> IO Int` do?

a. No function of this type can be defined.
b. Return a constant value.
c. Run the IO operation it has been given and return its value.
d. Query the user for a number and return it.

Which of these can a function of type `IO Int -> Int` do?

a. No function of this type can be defined.
b. Return a constant value.
c. Run the IO operation it has been given and return its value.
d. Query the user for a number and return it.

11.12 Exercises

- Set11a - basic IO exercises

- Set11b - advanced IO exercises

12 Lecture 12: fmap fmap fmap

12.1 Contents

- Functors

12.2 Functors

12.2.1 Preserving Structure

Remember the `map` function for lists? Here's the definition again:

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map g (x:xs) = g x : map g xs
```

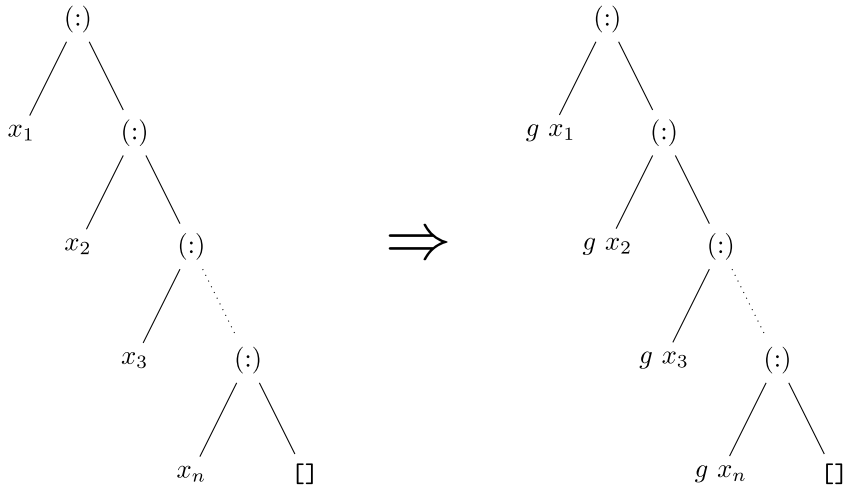
It applies a function `g :: a -> b` to each element of a list of type `[a]`, returning a list of type `[b]`. Another way to express the type of `map` would be `(a -> b) -> ([a] -> [b])`. This is the same type because `->` associates to right. The extra parentheses emphasize the fact that `map` converts the function `g :: a -> b` into a function `map g :: [a] -> [b]`. This means that `map` is a *higher-order function* that transforms functions to functions.

As `map` is parametrically polymorphic, its definition doesn't depend on the type of values stored in the list. Thus, every function of type `a -> b` is converted into a function of type `[a] -> [b]` using exactly the same logic. Using the definition above, we can see that:

```
map (|| True) [True, True, False]
  ==> [True || True, True || True, False || True]
  ==> [True, True, True]

map (+1) [1,2,3]
  ==> [1 + 1, 2 + 1, 3 + 1]
  ==> [2, 3, 4]

map (++"1") ["1", "2", "3"]
```

```
==> ["1" ++ "1", "2" ++ "1", "3" ++ "1"]
==> ["11", "21", "31"]
```

What's notable here is that `map` *preserves the structure* of a list. The length of the list and the relative positions of the elements are the same. The general idea is demonstrated in the picture below.



Mapping a function g to a list

Let's see if we can find other similar functions. A value of type `Maybe a` is kind of like a list of length at most 1. Let's map over a `Maybe`! Can you see the similarity with the definition of `map`?

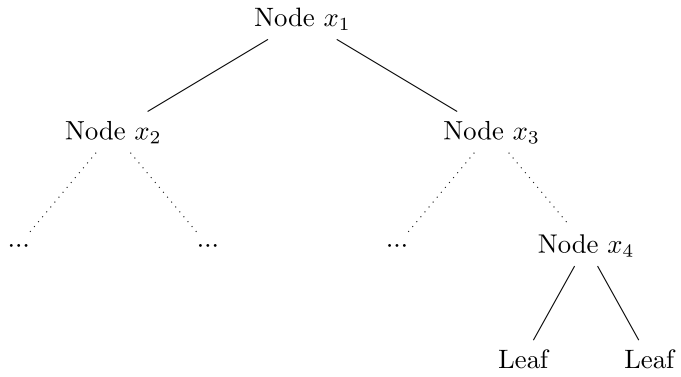
```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
```

Here too, the structure of the value is preserved. A `Nothing` turns into a `Nothing`, and a `Just` turns into a `Just`. Here too, we can think of the type as `(a -> b) -> (Maybe a -> Maybe b)`, converting (or "lifting") a normal function into a function that works on Maybes.

One more example: consider binary trees.

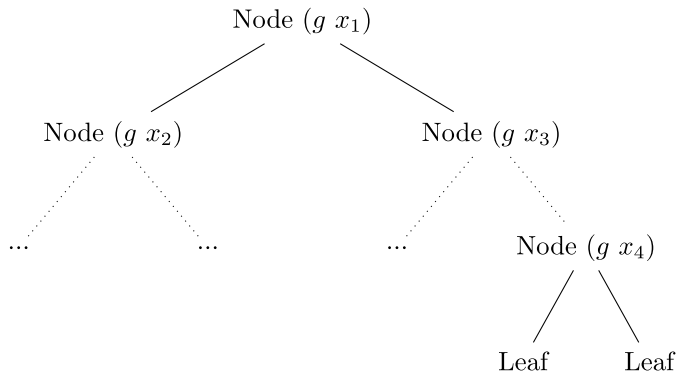
```
data Tree a = Leaf | Node a (Tree a) (Tree a)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Leaf = Leaf
mapTree f (Node val left right) = Node (f val) (mapTree f left)
         (mapTree f right)
```

A binary tree might look like this:



A binary tree

After `mapTree g` the tree would looke like this:



A binary tree after mapping `g` into its nodes

12.2.2 The `Functor` Class

Now we have three different structure-preserving mapping functions. Three similar operations over different types. Could we write a type class to capture this similarity?

```
map      :: (a -> b) ->      [a] ->      [b]
mapMaybe :: (a -> b) -> Maybe a  -> Maybe b
mapTree  :: (a -> b) -> Tree  a  -> Tree  b
```

A naive attempt at writing a type class runs into problems. If we try to abstract over `Maybe c`, it seems we can't write the right type for the map operation. We'd need to be able to change the type parameter `c` somehow.

```
class Mappable m where
  mapThing :: (a -> b) -> m -> m

instance Mappable (Maybe c) where
  mapThing :: (a -> b) -> Maybe c -> Maybe c
  mapThing = ...
```

Luckily Haskell type classes have a feature we haven't covered before. You can write classes for *type constructors* in addition to types. What does this mean? Let's just have a look at the standard type class `Functor` that does what we tried to do with our `Mappable`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Note how the type parameter `f` is a type constructor: it's being passed `a` and `b` arguments in different parts of the type of `fmap`. Now let's see the instance for `Maybe`.

```
instance Functor Maybe where
  -- In this instance, the type of fmap is:
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Now `fmap` has the right type and we can implement it like `mapMaybe`! Note how we've declared `instance Functor Maybe` instead of `instance Functor (Maybe a)`. The type `Maybe a` isn't a functor, the type constructor `Maybe` is.

The type constructor for lists is written `[]`. It's special syntax, just like other list syntax. However if the type `[a]` was written `List a`, the type constructor `[]` would mean `List`.

```
instance Functor [] where
  fmap = map
```

Here's the final of our examples, as a `Functor` instance.

```
data Tree a = Leaf | Node a (Tree a) (Tree a)

instance Functor Tree where
  fmap _ Leaf = Leaf
  fmap f (Node val left right) = Node (f val) (fmap f left) (fmap f
         right)
```

Sidenote: the term functor comes originally from a branch of mathematics called category theory. However, to work with Haskell you don't need to know any category theory. As you progress in learning Haskell, you might get interested in

category theory, and it can be a valuable source of new ideas for programming. Category theory can feel intimidating, so it's good to know that you can get on fine without it. For now, when you see functor, you can just think "something I can map over", or perhaps "a container".

Let's zoom out a bit. When we have an instance `Functor MyFun` we know that we can map a type `X` into a new type `MyFun X` (since `MyFun` is a type constructor), but also that we can lift a function `f` that takes an `X` argument into a function `fmap f` that takes a `MyFun X` argument! So you could say we're mapping both on the type level and the value level.

Oh right, one more thing. Once you've gotten the hang of `fmap` you might find yourself using it quite a bit. For code that uses `fmap` heavily it can be nice to use its infix alias, `<$>`. Consider the symmetry between `$` and `<$>` in these examples:

```
(+1) <$> [1,2,3]    ==>  [2,3,4]
not <$> Just False  ==>  Just True

reverse . tail  $       "hello"       ==>  "olle"
reverse . tail <$> Just "hello"       ==>  Just "olle"
-- which is the same as
fmap (reverse . tail) (Just "hello")  ==>  Just "olle"
```

12.3 Lawful Instances

What is this "preserving of the structure" that was mentioned above exactly? The following two *functor laws* are expected to hold for any `Functor` instance `f` (though unfortunately Haskell compilers can't enforce them):

1. `fmap id      === id`
2. `fmap (f . g) === fmap f . fmap g`

Don't worry if that sounded abstract! The first law says that a functor maps `id :: a -> a` into `id :: f a -> f a`. (`id` is the identity function, meaning that `id x = x`.) Let's be concrete and see how it works for the list `[1,2,3]`:

```
fmap id [1,2,3] ==> map id [1,2,3]
                ==> map id (1:[2,3])
                ==> id 1 : map id [2,3]
                ==> 1 : map id [2,3]
                ==> 1 : id 2 : map id [3]
                ==> 1 : 2 : id 3 : map id []
                ==> 1 : 2 : 3 : []
                === [1,2,3]
```

On the other hand,

```
id [1,2,3] ==> [1,2,3]
```

Hence, the result of `fmap id [1,2,3]` was the same as the result of `id [1,2,3]`, so the first functor law holds in this case. It's not hard to show that the first functor law holds for any list whatsoever.

The first functor law is really a very simple proposition if you think about it. It just says that if we apply `fmap` to a function that changes nothing (`id`) the resulting function (`fmap id`) again changes nothing. Thus, the act of applying `fmap` itself preserves the structure of the functor.

How about the second functor law? For lists, consider what happens if we `fmap` the function `negate.(*2)` (remember, `negate` maps `x` to `-x` and `(*2)` multiplies its argument by `2`):

```
fmap (negate.(*2)) [1,2,3] ==> map (negate.(*2)) [1,2,3]
                           ==> (negate.(*2)) 1 : map (negate.(*2))
        [2,3]
                           ==> negate (1 * 2)  : map (negate.(*2))
        [2,3]
                           ==> -2 : map (negate.(*2)) [2,3]
                           ==> -2 : (negate.(*2)) 2 : map (negate.
        (*2)) [3]
                           ==> -2 : -4 : map (negate.(*2)) [3]
                           ==> -2 : -4 : (negate.(*2)) 3 : map
        (negate.(*2)) []
                           ==> -2 : -4 : -6 : []
                           ==> [-2,-4,-6]
```

Let's consider the right-hand side of the second functor law in this case:

```
(fmap negate . fmap (*2)) [1,2,3] ==> (map negate . map (*2))
        [1,2,3]
                                      ==> map negate (map (*2) [1,2,3])
                                      ==> map negate [2,4,6]
                                      ==> [-2,-4,-6]
```

The second functor law turns out to hold in this particular case. In fact, it holds in all cases (exercise!).

In general, the second functor law says that composing two functions first and then applying `fmap` must yield the same result as performing `fmap` on those functions and

then composing the resulting function. In other words, the order of applying `fmap` and composing doesn't matter. (These two operations are said to *commute*.)

There are also higher-order functions that fail to satisfy functor laws. Consider the function `badMap`:

```
badMap :: (a -> b) -> [a] -> [b]
badMap f [] = []
badMap f (x:y:xs) = f x : badMap f xs
badMap f (x:xs) = f x : badMap f xs
```

This function violates the first functor law. For instance:

```
badMap id [1,2,3] ==> badMap id (1:2:[3])
                  ==> id 1 : badMap id [3]
                  ==> 1 : badMap id [3]
                  ==> 1 : badMap id (3:[])
                  ==> 1 : id 3 : badMap id []
                  ==> 1 : 3 : []
                  ==> [1,3]
```

Applying `badMap id` to the list `[1,2,3]` changed the list as the element `2` was dropped.

As mentioned, Haskell compilers can't detect if a functor obeys its laws or not. A Haskell compiler would happily accept an instance `Functor []` that used `badMap` instead of `map` as the `fmap` implementation. This is a limitation of the type system of Haskell. There are technologies such as LiquidHaskell or dependently typed languages such as Agda, Idris, Coq, or Lean that could actually enforce functor laws so that unlawful functor instances wouldn't compile. These technologies are outside the scope of this course, however.

12.4 Sidenote: Kinds

Remember that `Functor` was a class for type constructors. If we try to define an instance of `Functor` for a type, we get an error:

```
Prelude> instance Functor Int where

<interactive>:1:18: error:
    • Expected kind ‘* -> *’, but ‘Int’ has kind ‘*’
    • In the first argument of ‘Functor’, namely ‘Int’
      In the instance declaration for ‘Functor Int’
```

The error message talks about *kinds*. Kinds are *types of types*. A type like `Int`, `Bool` or `Maybe Int` that can contain values has kind `*`. A type constructor has a kind that looks like a function, for example, `Maybe` has kind `* -> *`. This means that the `Maybe` type constructor must be applied to a type of kind `*` to get a type of kind `*`.

We can ask GHCi for the kinds of types:

```
Prelude> :kind Int
Int :: *
Prelude> :kind Maybe
Maybe :: * -> *
Prelude> :kind Maybe Int
Maybe Int :: *
```

If we ask GHCi for info about the `Functor` class, it tells us that instances of `Functor` must have kind `* -> *`:

```
Prelude> :info Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
...
```

Here are some examples of even more complex kinds.

```
-- multiple type parameters
Prelude> :kind Either
Either :: * -> * -> *
Prelude> data Either3 a b c = Left a | Middle b | Right c
Prelude> :kind Either3
Either3 :: * -> * -> * -> *
-- a type parameter of kind *->*
Prelude> data IntInside f = IntInside (f Int)
Prelude> :kind IntInside
IntInside :: (* -> *) -> *
```

You won't bump into kinds that much in Haskell programming, but sometimes you'll see error messages that talk about kinds, so it's good to know what they are.

12.5 `Foldable`, Again

We briefly covered the class `Foldable`, which occurs in many type signatures of basic functions, in part 1. For example:

```haskell
length :: Foldable t => t a -> Int
sum :: (Foldable t, Num a) => t a -> a
minimum :: (Foldable t, Ord a) => t a -> a
foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m
```

From these type signatures we can see that `Foldable`, just like `Functor`, is a class for type constructors (things of kind `* -> *`). The essence of `Foldable` is to be a class for *things you can fold over*. The class definition could be as simple as

```haskell
class Foldable (t :: *->*) where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

However, for performance reasons, the class contains many methods (you can see them yourself by checking `:info Foldable` in GHCi!), but when we're defining an instance for `Foldable` it's enought to define just `foldr`.

Another way of thinking of the `Foldable` class is processing elements *left-to-right*, in other words, if `Functor` was the class for containers, `Foldable` is the class for *ordered containers*.

As an example, let's implement `Functor` and `Foldable` for our own pair type.

```haskell
data Pair a = Pair a a
  deriving Show

instance Functor Pair where
  -- fmap f applies f to all values
  fmap f (Pair x y) = Pair (f x) (f y)

instance Foldable Pair where
  -- just like applying foldr over a list of length 2
  foldr f initialValue (Pair x y) = f x (f y initialValue)

-- an example function that uses both instances
doubleAndCount :: (Functor f, Foldable f) => f Int -> Int
doubleAndCount = sum . fmap (*2)
```

Now, we can use `Pair` almost wherever we can use a list:

```haskell
fmap (+1) (Pair 3 6)   ==> Pair 4 7
fmap (+1) [3,6]        ==> [4,7]

foldr (*) 1 (Pair 3 6) ==> 18
```

```
foldr (*) 1 [3,6]       ==> 18

length (Pair 3 6)       ==> 2
length [3,6]            ==> 2

minimum (Pair 3 6)      ==> 3
minimum [3,6]           ==> 3

doubleAndCount (Pair 3 6)  ==> 18
doubleAndCount [3,6]       ==> 18
```

Other types we've met that are `Foldable` include `Data.Map` and `Data.Array`.

12.6 Recap

So, to summarize, a functor is a type constructor `f` and the corresponding `Functor f` instance such that `fmap` satisfies the two functor laws. These laws assert that `fmap` must preserve the identity function and distribute over function composition. More informally, `fmap` lifts a function `g :: a -> b` operating on values to one operating on containers: `fmap g :: f a -> f b`. Basically all well-behaving data structures in Haskell are functors.

12.7 Quiz

What's the type of `fmap`?

a. `a -> b -> f a -> f b`
b. `(a -> b) -> f a -> f b`
c. `Functor f => a -> b -> f a -> f b`
d. `Functor f => (a -> b) -> f a -> f b`

Which code snippet completes the next `Functor` instance?

```
data Container x = Things x [x]

instance Functor Container where
  ????
```

a. `fmap f (Things x ys) = Things (f x) [f x]`
b. `fmap f (Things x ys) = Things (f x) (map f ys)`
c. `fmap f (Things x ys) = Things (f x) ys`
d. `fmap f (Things x ys) = f (Things x ys)`

What's the kind of `[a]`?

a. *
b. * -> *
c. [a]

What's the kind of `Foo`?

```
data Foo x = FooConst
```

a. *
b. * -> *
c. Foo

What's the kind of `Bar`?

```
data Bar = Baz | Qux Int
```

a. *
b. * -> *
c. Bar

What is the value of `foldr (-) 1 (Just 2)`?

a. -1
b. 1
c. Just -1
d. Just 1

Which code snippet completes the next `Foldable` instance?

```
data Container x = Things x [x]

instance Foldable Container where
  ????
```

a. foldr f z (Things x ys) = f x z
b. foldr f z (Things x ys) = foldr f x ys
c. foldr f z (Things x ys) = f x (foldr f z ys)
d. foldr f z (Things x ys) = foldr f z (x:ys)

12.8 Exercises

- Set12

13 Lecture 13: A Monoid in the Category of Problems

- Monads

In this lecture we'll build up to the concept of a *monad* using a number of examples. By now you should be familiar with all the Haskell features needed for understanding monads.

Monads are a famously hard topic in programming, which is partly due to weird terminology, partly due to bad tutorials, and partly due to trying to understand monads too early when learning Haskell. Monads are introduced this late in the course in an attempt to make understanding them easier.

If you find this lecture hard, don't despair, many others have found the topic hard as well. There are many many productive Haskell programmers who have managed to understand monads, so the task is not hopeless.

One final word of caution: monads, like functors, are a concept originally from a branch of mathematics called Category Theory. However, and I can't stress this enough, *you do not need to know anything or even care about category theory to understand monads in Haskell programming.* Just like one can work with object-oriented programming or functional programming without knowing the theory of objects or functions, one can work with monads without understanding the math associated with them. Category theory can be a rewarding topic for a functional programmer, but it's not a mandatory one.

13.1 Example 1: Maybes

When working with many `Maybe` values, the code tends to become a bit messy. Let's look at some examples. First, we combine some functions returning `Maybe String`. Note the nested `case` we need in `stealSecret`: It's not fun to write.

```
-- Try to login with a password.
-- `Just username` on success, `Nothing` otherwise.
login :: String -> Maybe String
login "f4bulous!" = Just "unicorn73"
login "swordfish" = Just "megahacker"
login _           = Nothing

-- Get a secret associated with a user.
-- Not all users have secrets.
secret :: String -> Maybe String
secret "megahacker" = Just "I like roses"
```

```
secret _            = Nothing

-- Login and return the user's secret, if any
stealSecret :: String -> Maybe String
stealSecret password =
  case login password of
    Nothing -> Nothing
    Just user -> case secret user of
                   Nothing -> Nothing
                   Just s -> Just ("Stole secret: "++s)



stealSecret "swordfish"  ==>  Just "Stole secret: I like roses"
stealSecret "f4bulous!"  ==>  Nothing
stealSecret "wrong_password"  ==>  Nothing
```

Next up, we modify a list of pairs. We use the `Maybe`-returning function `lookup` from the Prelude. Here we have an if inside a case instead of a nested case.

```
-- Get the value corresponding to a key from a key-value list.
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b



-- Set the value of key to val in the given key-value list,
-- but only if val is larger than the current value!
increase :: Eq a => a -> Int -> [(a,Int)] -> Maybe [(a,Int)]
increase key val assocs =
  case lookup key assocs
  of Nothing -> Nothing
     Just x -> if (val < x)
                 then Nothing
                 else Just ((key,val) : delete (key,x) assocs)
```

This type of code is pretty common, and usually repeats the same pattern: if any intermediate result is `Nothing`, the whole result is `Nothing`. Let's try to make writing code like this easier by defining a *chaining operator* `?>`. The chaining operator takes a result and the next step of computation, and only runs the next step if the result was a `Just` value.

```
(?>) :: Maybe a -> (a -> Maybe b) -> Maybe b
-- if we failed, don't even bother running the next step:
Nothing ?> _ = Nothing
```

```
-- otherwise run the next step:
Just x  ?> f = f x
```

The chaining operator streamlines our examples nicely. Note how we can define simple helper functions that take care of one step of the computation instead of writing one big expression.

```
stealSecret :: String -> Maybe String
stealSecret password =
    login password ?>
    secret ?>
    decorate
  where decorate s = Just ("Stole secret: "++s)
```

```
increase :: Eq a => a -> Int -> [(a,Int)] -> Maybe [(a,Int)]
increase key val assocs =
    lookup key assocs ?>
    check ?>
    buildResult
  where check x
          | val < x   = Nothing
          | otherwise = Just x
        buildResult x = Just ((key,val) : delete (key,x) assocs)
```

Here's another example: safe list indexing built from `safeHead` and `safeTail`:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x

safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (x:xs) = Just xs

safeThird :: [a] -> Maybe a
safeThird xs = safeTail xs ?> safeTail ?> safeHead

safeNth :: Int -> [a] -> Maybe a
safeNth 0 xs = safeHead xs
safeNth n xs = safeTail xs ?> safeNth (n-1)
```

```
safeThird [1,2,3,4]
  ==> Just 3
```

```
safeThird [1,2]
  ==> Nothing
safeNth 5 [1..10]
  ==> Just 6
safeNth 11 [1..10]
  ==> Nothing
```

PS. note that `?>` *associates to the left* as is the default in Haskell. That means that `op ?> f ?> g` means `(op ?> f) ?> g`. The alternative, `op ?> (f ?> g)` would not even type check!

Sidenote: this `?>` operator expresses the if-result pattern that's very common in other languages. Here is how one would write `op val ?> f` in Python and Java.

```
# Python
x = op(val)
if x:
  f(x)
```

```
// Java
Object x = op(val);
if (x != null) {
  f(x);
}
```

The difference between the if-result pattern and our `?>` is that we use the `Nothing` value to explicitly signal failure, instead of relying on the fact that any variable can be `None` (or `False`) in Python, or that any `Object` reference can be `null` in Java.

13.2 Example 2: Logging

Let's explore the concept of chaining with another example: logging. The type `Logger` represents a value plus a list of log messages (produced by the computation that produced the value).

```
-- Logger definition
data Logger a = Logger [String] a  deriving Show

getVal :: Logger a -> a
getVal (Logger _ a) = a
getLog :: Logger a -> [String]
getLog (Logger s _) = s
```

```
-- Primitive operations:
nomsg :: a -> Logger a
nomsg x = Logger [] x        -- a value, no message

annotate :: String -> a -> Logger a
annotate s x = Logger [s] x  -- a value and a message

msg :: String -> Logger ()
msg s = Logger [s] ()        -- just a message
```

Here's a `login` function that logs some details about the usernames and passwords it processes. Note how we run into complicated code in `login` when we need to handle multiple `Logger` values.

```
validateUser :: String -> Logger Bool
validateUser "paul.atreides" = annotate "Valid user" True
validateUser "ninja" = nomsg True
validateUser u = annotate ("Invalid user: "++u) False

checkPassword :: String -> String -> Logger Bool
checkPassword "paul.atreides" "muad'dib" = annotate "Password ok"
         True
checkPassword "ninja"         ""         = annotate "Password ok"
         True
checkPassword _               pass       = annotate ("Password
         wrong: "++pass) False

login :: String -> String -> Logger Bool
login user password =
  let validation = validateUser user
  in if (getVal validation)
       then let check = checkPassword user password
            in Logger (getLog validation ++ getLog check) (getVal
         check)
       else validation
```

```
login "paul.atreides" "muad'dib"
  ==> Logger ["Valid user","Password ok"] True
login "paul.atreides" "arrakis"
  ==> Logger ["Valid user","Password wrong: arrakis"] False
login "ninja" ""
  ==> Logger ["Password ok"] True
login "leto.atreides" "paul"
  ==> Logger ["Invalid user: leto.atreides"] False
```

Let's try to streamline this code by defining a chaining operator for `Logger`. The important thing when doing multiple `Logger` operations is to preserve all the logs. Here's a chaining operator, `#>`, and an example of how it can be used to log some arithmetic computations.

```
(#>) :: Logger a -> (a -> Logger b) -> Logger b
Logger la a #> f = let Logger lb b = f a  -- feed value to next step
                   in Logger (la++lb) b   -- bundle result with all
        messages
```

```
-- square a number and log a message about it
square :: Int -> Logger Int
square val = annotate (show val ++ "^2") (val^2)

-- add 1 to a number and log a message about it
add :: Int -> Logger Int
add val = annotate (show val ++ "+1") (val+1)

-- double a number and log a message about it
double :: Int -> Logger Int
double val = annotate (show val ++ "*2") (val*2)

-- compute the expression 2*(x^2+1) with logging
compute :: Int -> Logger Int
compute x =
    square x
    #> add
    #> double
```

```
compute 3
  ==> Logger ["3^2","9+1","10*2"] 20
```

We can streamline `login` quite a bit by using `#>`. Note how we don't need to worry about combining logs together. Also note how we use a lambda expression instead of defining a helper function.

```
login :: String -> String -> Logger Bool
login user password =
  validateUser user
  #>
  \valid -> if valid then checkPassword user password
                     else nomsg False
```

To ramp things up a bit, let's use `Logger` in a recursive list processing function. Here's a logging version of `filter`. Note how the code chains a log message before the recursive call in order to keep the order of log entries nice.

```
-- sometimes you don't need the previous value:
(##>) :: Logger a -> Logger b -> Logger b
Logger la _ ##> Logger lb b = Logger (la++lb) b

filterLog :: (Eq a, Show a) => (a -> Bool) -> [a] -> Logger [a]
filterLog f [] = nomsg []
filterLog f (x:xs)
   | f x       = msg ("keeping "++show x) ##> filterLog f xs #>
         (\xs' -> nomsg (x:xs'))
   | otherwise = msg ("dropping "++show x) ##> filterLog f xs
```

```
filterLog (>0) [1,-2,3,-4,0]
  ==> Logger ["keeping 1","dropping -2","keeping 3","dropping
        -4","dropping 0"] [1,3]
```

13.3 Example 3: Keeping State

In the previous example we just wrote some state (the log). Sometimes we need computations that change some sort of shared state. Let's look at accounts in a small bank. We'll first define a datatype for the state of the bank: the balances of all accounts, as a map from account name to balance.

```
import qualified Data.Map as Map

data Bank = Bank (Map.Map String Int)
  deriving Show
```

Here's how we can deposit some money to an account. We use the function `adjust` from `Data.Map` to modify the map.

```
-- Apply a function to one value in a map
Map.adjust :: Ord k => (a -> a) -> k -> Map.Map k a -> Map.Map k a
```

```
deposit :: String -> Int -> Bank -> Bank
deposit accountName amount (Bank accounts) =
```

```
  Bank (Map.adjust (\x -> x+amount) accountName accounts)
```

Withdrawing money is a bit more complicated, since we want to handle some special cases like the account not existing, or the account not having enough money. We use the library function `findWithDefault` to help us along.

```
-- Fetch the value corresponding to a key from a map,
-- or a default value in case the key does not exist
Map.findWithDefault :: Ord k => a -> k -> Map.Map k a -> a
```

```
withdraw :: String -> Int -> Bank -> (Int,Bank)
withdraw accountName amount (Bank accounts) =
  let -- balance is 0 for a nonexistant account
      balance = Map.findWithDefault 0 accountName accounts
      -- can't withdraw over balance
      withdrawal = min amount balance
      newAccounts = Map.adjust (\x -> x-withdrawal) accountName
         accounts
  in (withdrawal, Bank newAccounts)
```

Finally, let's write a function that takes at most 100 money from one account, splits the money in half, and deposits it in two accounts. Pay attention to how we need to carefully thread the different versions of the bank, `bank`, `bank1`, `bank2` and `bank3` to make sure all transactions happen in the right order.

```
share :: String -> String -> String -> Bank -> Bank
share from to1 to2 bank =
  let (amount,bank1) = withdraw from 100 bank
      half = div amount 2
      -- carefully preserve all money, even if amount was an odd
         number
      rest = amount-half
      bank2 = deposit to1 half bank1
      bank3 = deposit to2 rest bank2
  in bank3
```

```
share "wotan" "siegfried" "brunhilde"
      (Bank (Map.fromList [("brunhilde",0),("siegfried",0),
         ("wotan",1000)]))
   ==> Bank (Map.fromList [("brunhilde",50),("siegfried",50),
         ("wotan",900)])
```

```
share "wotan" "siegfried" "brunhilde"
      (Bank (Map.fromList [("brunhilde",0),("siegfried",0),
         ("wotan",91)]))
   ==> Bank (Map.fromList [("brunhilde",46),("siegfried",45),
         ("wotan",0)])
```

Code like this turns up often in Haskell when you're doing serial updates to one value, while also performing some other computations on the side. It's easy to make a mistake, and the type system won't help you if you e.g. reuse the `bank1` value. Let's rewrite `share` so that we don't need to refer to the bank itself. We can again use the same chaining idea to accomplish this.

```
-- `BankOp a` is an operation that transforms a Bank value,
-- while returning a value of type `a`
data BankOp a = BankOp (Bank -> (a,Bank))

-- running a BankOp on a Bank
runBankOp :: BankOp a -> Bank -> (a,Bank)
runBankOp (BankOp f) bank = f bank

-- Running one BankOp after another
(+>>) :: BankOp a -> BankOp b -> BankOp b
op1 +>> op2 = BankOp combined
  where combined bank = let (_,bank1) = runBankOp op1 bank
                        in runBankOp op2 bank1

-- Running a parameterized BankOp, using the value returned
-- by a previous BankOp.  The implementation is a bit tricky
-- but it's enough to understand how +> is used for now.
(+>) :: BankOp a -> (a -> BankOp b) -> BankOp b
op +> parameterized = BankOp combined
  where combined bank = let (a,bank1) = runBankOp op bank
                        in runBankOp (parameterized a) bank1

-- Make a BankOp out of deposit.
-- There is no return value so we use ().
depositOp :: String -> Int -> BankOp ()
depositOp accountName amount = BankOp depositHelper
  where depositHelper bank = ((), deposit accountName amount bank)

-- Make a BankOp out of withdraw. Note how
--   withdraw accountName amount :: Bank -> (Int,Bank)
-- is almost a BankOp already!
withdrawOp :: String -> Int -> BankOp Int
withdrawOp accountName amount = BankOp (withdraw accountName amount)
```

Let's see how chaining works with these bank operations.

```
Prelude> bank = Bank (Map.fromList [("edsger",10),("grace",50)])

-- Running a number of operations using +>>

Prelude> runBankOp (depositOp "edsger" 1) bank
((),Bank (fromList [("edsger",11),("grace",50)]))

Prelude> runBankOp (depositOp "edsger" 1 +>> depositOp "grace" 1)
         bank
((),Bank (fromList [("edsger",11),("grace",51)]))

Prelude> runBankOp (depositOp "edsger" 1 +>> depositOp "grace" 1 +>>
         withdrawOp "edsger" 11) bank
(11,Bank (fromList [("edsger",0),("grace",51)]))

-- Using +> to implement a transfer from one account to the other:

Prelude> runBankOp (withdrawOp "edsger" 5 +> depositOp "grace") bank
((),Bank (fromList [("edsger",5),("grace",55)]))

Prelude> runBankOp (withdrawOp "edsger" 100 +> depositOp "grace")
         bank
((),Bank (fromList [("edsger",0),("grace",60)]))
```

Note how a value of type `BankOp` represents a process that transforms the bank. The initial state of the bank must be supplied using `runBankOp`. This makes sense because `BankOp` transformations can be composed, unlike `Bank` states. Having to use `runBankOp` makes the distinction between *defining* operations and *executing them* clearer.

Now that we're familiar with manipulating `BankOp` values, we can implement `share` as a `BankOp`. We implement a helper `distributeOp` to make the code a bit neater.

```
-- distribute amount to two accounts
distributeOp :: String -> String -> Int -> BankOp ()
distributeOp to1 to2 amount =
  depositOp to1 half
  +>>
  depositOp to2 rest
  where half = div amount 2
        rest = amount - half

shareOp :: String -> String -> String -> BankOp ()
shareOp from to1 to2 =
  withdrawOp from 100
  +>
  distributeOp to1 to2
```

```
runBankOp (shareOp "wotan" "siegfried" "brunhilde")
          (Bank (Map.fromList [("brunhilde",0),("siegfried",0),
          ("wotan",1000)]))
  ==> ((),Bank (Map.fromList [("brunhilde",50),("siegfried",50),
          ("wotan",900)]))

runBankOp (shareOp "wotan" "siegfried" "brunhilde")
          (Bank (Map.fromList [("brunhilde",0),("siegfried",0),
          ("wotan",91)]))
  ==> ((),Bank (Map.fromList [("brunhilde",46),("siegfried",45),
          ("wotan",0)]))
```

That was pretty clean wasn't it? We don't need to mention the bank at all, we can almost program as if in an imperative language while staying completely pure.

You can find all of this code in the course repository under `exercises/Examples/Bank.hs`.

13.4 Finally: The Monad Type Class

We've now seen three different types with a chaining operation:

```
(?>) :: Maybe a -> (a -> Maybe b) -> Maybe b
(#>) :: Logger a -> (a -> Logger b) -> Logger b
(+>) :: BankOp a -> (a -> BankOp b) -> BankOp b
```

Just like previously with `map` and `Functor`, there is a type class that captures this pattern. Note that `Monad` is a class for *type constructors*, just like `Functor`.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

There are some additional operations in `Monad` too:

```
  -- lift a normal value into the monad
  return :: a -> m a
  -- simpler chaining (like our ##>)
  (>>) :: m a -> m b -> m b
  a >> b  =  a >>= \_ -> b      -- remember: _ means ignored argument
```

Recall that the `Functor` class was about a generic `map` operation. Similarly, the `Monad` class is just about a generic chaining operation.

```
fmap :: Functor f => (a->b) -> f a -> f b
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

The expression `operation >>= next` takes a monadic operation `operation :: m a`, and does some further computation with the value that it produces using `next :: a -> m b`. If this feels too abstract, just recall how chaining works for `Maybe`:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
-- if we failed, don't even bother running the next step
Nothing >>= _ = Nothing
-- otherwise run the next step
Just x  >>= f = f x
```

13.5 Maybe is a Monad!

Here's the full `Monad` instance for `Maybe` and some examples.

```
instance  Monad Maybe  where
    (Just x) >>= k      = k x
    Nothing  >>= _      = Nothing

    (Just _) >>  k      = k
    Nothing  >>  _      = Nothing

    return x            = Just x
```

```
Just 1 >>= \x -> return (x+1)
  ==> Just 2
Just "HELLO" >>= (\x -> return (length x)) >>= (\x -> return (x+1))
  ==> Just 6
Just "HELLO" >>= \x -> Nothing
  ==> Nothing
Just "HELLO" >> Just 2
  ==> Just 2
Just 2 >> Nothing
  ==> Nothing
```

Here are the `stealSecret` and `increase` examples rewritten with monad operations. The changes are `?>` to `>>=` and `Just` to `return`.

```
stealSecret :: String -> Maybe String
stealSecret password =
    login password >>=
    secret >>=
    decorate
  where decorate s = return ("Stole secret: "++s)
```

```
-- Set the value of key to val in the given key-value list,
-- but only if val is larger than the current value!
increase :: Eq a => a -> Int -> [(a,Int)] -> Maybe [(a,Int)]
increase key val assocs =
    lookup key assocs >>=
    check >>=
    buildResult
  where check x
          | val < x   = Nothing
          | otherwise = return x
        buildResult x = return ((key,val) : delete (key,x) assocs)
```

13.6 The Return of do

Here's an example of what a complex monad operation might look like.

```
f = op1 >>= continue
  where continue  x   = op2 >> op3 >>= continue2 x
        continue2 x y = op4 >> op5 x y
```

Let's see what happens when we transform this code a bit. First off, let's inline the definitions.

```
f = op1 >>= (\x ->
               op2 >>
               op3 >>= (\y ->
                          op4 >>
                          op5 x y))
```

Due to lambda expressions continuing to the end of the expression, we can omit the parentheses. Let's also indent differently.

```
f = op1 >>= \x ->
    op2 >>
    op3 >>= \y ->
    op4 >>
    op5 x y
```

Now we can notice the similarity with `do` notation. The `do` block below is actually the same code!

```
f = do x <- op1
       op2
       y <- op3
       op4
       op5 x y
```

To clarify, `do` notation is just a nicer syntax for the monad operations (`>>=` and `>>`) and lambdas. Here's how do notation gets transformed into monad operations. Note! the definition is recursive.

```
do x <- op a        ~~~>        op a >>= \x -> do ...
   ...


do op a             ~~~>        op a >> do ...
   ...


do let x = expr     ~~~>        let x = expr in do ...
   ...


do finalOp          ~~~>        finalOp
```

Here's `safeNth` using do notation:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x

safeTail :: [a] -> Maybe [a]
```

```haskell
safeTail [] = Nothing
safeTail (x:xs) = Just xs

safeNth :: Int -> [a] -> Maybe a
safeNth 0 xs = safeHead xs
safeNth n xs = do t <- safeTail xs
                  safeNth (n-1) t
```

Here is `increase` one last time, now with do notation

```haskell
-- Set the value of key to val in the given key-value list,
-- but only if val is larger than the current value!
increase :: Eq a => a -> Int -> [(a,Int)] -> Maybe [(a,Int)]
increase key val assocs =
  do oldVal <- lookup key assocs
     check oldVal
     return ((key,val) : delete (key,oldVal) assocs)
  where check x
          | val < x   = Nothing
          | otherwise = return x
```

13.7 Logger is a Monad!

We should be able to write a `Monad` instance for `Logger` ourselves, by setting `>>=` to `#>`. However, due to recent changes in the Haskell language we must implement `Functor` and `Applicative` instances to be allowed to implement the `Monad` instance. `Functor` we've already met, but what's `Applicative`? We'll find out later. Let's implement the instances:

```haskell
import Control.Monad

data Logger a = Logger [String] a  deriving Show

msg :: String -> Logger ()
msg s = Logger [s] ()

-- The Functor instance just maps over the stored value
instance Functor Logger where
  fmap f (Logger log x) = Logger log (f x)

-- This is an Applicative instance that works for any
-- monad, you can just ignore it for now. We'll get back
-- to Applicative later.
instance Applicative Logger where
  pure = return
  (<*>) = ap
```

```haskell
-- Finally, the Monad instance
instance Monad Logger where
  return x = Logger [] x
  Logger la a >>= f = Logger (la++lb) b
    where Logger lb b = f a
```

We don't need the `nomsg` operation any more since it's just `return`. We can also reimplement the `annotate` operation using monad operations.

```haskell
nomsg :: a -> Logger a
nomsg x = return x

annotate :: String -> a -> Logger a
annotate s x = msg s >> return x
```

Here are the `compute` and `filterLog` examples rewritten using do-notation. Note how nice `filterLog` is with do-notation.

```haskell
compute x = do
  a <- annotate "^2" (x*x)
  b <- annotate "+1" (a+1)
  annotate "*2" (b*2)

filterLog :: (Show a) => (a -> Bool) -> [a] -> Logger [a]
filterLog f [] = return []
filterLog f (x:xs)
   | f x       = do msg ("keeping "++show x)
                    xs' <- filterLog f xs
                    return (x:xs')
   | otherwise = do msg ("dropping "++show x)
                    filterLog f xs
```

```haskell
compute 3
  ==> Logger ["^2","+1","*2"] 20
filterLog (>0) [1,-2,3,-4,0]
  ==> Logger ["keeping 1","dropping -2","keeping 3","dropping
         -4","dropping 0"] [1,3]
```

13.8 The State Monad

Haskell's `State` monad is a generalized version of our `BankOp` type. The `State` type is parameterized by two types, the first being the type of the state, and the second the type of the value produced. `State Bank a` would be equivalent to our `BankOp a`. You can find the `State` monad in the module `Control.Monad.Trans.State` of the `transformers` package. Here's a simplified implementation of `State`.

```
data State s a = State (s -> (a,s))

runState (State f) s = f s

-- operation that overwrites the state (and produces ())
put :: s -> State s ()
put state = State (\oldState -> ((),state))

-- operation that produces the current state
get :: State s s
get = State (\state -> (state,state))

-- operation that modifies the current state with a function (and
        produces ())
modify :: (s -> s) -> State s ()
modify f = State (\state -> ((), f state))

-- Functor and Applicative instances skipped

instance Monad (State s) where
  return x = State (\s -> (x,s))

  op >>= f = State h
    where h state0 = let (val,state1) = runState op state0
                         op2 = f val
                     in runState op2 state1
```

Note how we declare an instance `Monad (State s)`. We're using a *partially-applied type constructor* because instances of `Monad` can only be declared for type constructors that take one more type parameter. This might be a bit clearer if you look at how `m`, `Maybe` and `State` occur in the type of `>>=` below.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b

instance Monad Maybe where
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

instance Monad (State s) where
  (>>=) :: State s a -> (a -> State s b) -> State s b
```

Let's look at some examples of working with `State`. To start off, let's consider computations of type `State Int a`, which represent working with a simple counter.

```
-- adds i to the value of the counter
add :: Int -> State Int ()
add i = do old <- get
           put (old+i)
```

```
runState (add 1 >> add 3 >> add 5 >> add 6) 0
  ==> ((),15)
```

```
example :: State Int Int
example = do add 3           -- increment state by 3
             value <- get    -- value is current state, i.e.
         initial+3
             add 1000        -- increment state by 1000
             put (value + 1) -- overwrite state with value+1, i.e.
         initial+4
             return value    -- produce value, i.e. intial+3
```

```
runState example 1
  ==> (4,5)           -- initial is 1, state is initial+4=5,
         produces initial+3=4
```

Note how a value of type `State s a` represents a process that transforms the state (just like `BankOp`). The initial state must be supplied using `runState`. Again, having to use `runState` makes the distinction between *defining* operations and *executing them* clearer.

A state can replace an accumulator parameter when processing a list. Here are two examples: finding the largest element of a list, and finding values in a list that occur directly after a `0`.

```
findLargest :: Ord a => [a] -> State a ()
findLargest [] = return ()
findLargest (x:xs) = do
  modify (\y -> max x y)  -- update state with max of current value
         and previous largest value
  findLargest xs          -- process rest of list
```

```
runState (findLargest [1,2,7,3]) 0  ==>  ((),7)
```

```
-- store the given value in the state list
remember :: a -> State [a] ()
remember x = modify (x:)

valuesAfterZero :: [Int] -> ((),[Int])
valuesAfterZero xs = runState (go xs) []
  where go :: [Int] -> State [Int] ()
        go (0:y:xs) = do remember y
                         go (y:xs)
        go (x:xs) = go xs
        go [] = return ()
```

```
valuesAfterZero [0,1,2,3,0,4,0,5,0,0,6]
  ==> ((),[6,0,5,4,1])
```

Note! By the way, the actual implementation of the `State` monad doesn't have a `State` constructor like our simplified exmaple. If you want to wrap a function into a `State` operation, use this helper instead:

```
state :: (s -> (a, s)) -> State s a
```

13.9 The Return of `mapM`

The control structures from the IO lecture work in *all monads*. Here are their real types.

```
when :: Monad m => Bool -> m () -> m ()        -- conditional
        operation
unless :: Monad m => Bool -> m () -> m ()      -- same, but
        condition is flipped
replicateM :: Monad m => Int -> m a -> m [a]   -- do something many
        times
replicateM_ :: Monad m => Int -> m a -> m ()   -- same, but ignore
        the results
mapM :: Monad m => (a -> m b) -> [a] -> m [b]  -- do something on a
        list's elements
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()  -- same, but ignore
        the results
```

```
forM  :: Monad m => [a] -> (a -> m b) -> m [b] -- mapM but arguments
         reversed
forM_ :: Monad m => [a] -> (a -> m b) -> m ()  -- same, but ignore
         the results
```

As we can see here, we can use `mapM` over all of the monads we've met so far:

```
mapM (\x -> if (x>0) then Just (x-1) else Nothing) [1,2,3]  ==>
         Just [0,1,2]
mapM (\x -> if (x>0) then Just (x-1) else Nothing) [1,0,3]  ==>
         Nothing

mapM (\x -> msg "increment" >> msg (show x) >> return (x+1)) [1,2,3]
  ==> Logger ["increment","1","increment","2","increment","3"]
         [2,3,4]

runState (mapM (\x -> modify (x+) >> return (x+1)) [1,2,3]) 0
  ==> ([2,3,4],6)
```

Some more examples:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
firsts :: [[a]] -> Maybe [a]
firsts xs = forM xs safeHead



firsts [[1,2,3],[4,5],[6]] ==> Just [1,4,6]
firsts [[1,2,3],[],[6]]    ==> Nothing



-- an abbreviated version of an example from the last section
findLargest :: Ord a => [a] -> State a ()
findLargest xs = mapM_ update xs
  where update x = modify (\y -> max x y)



runState (findLargest [1,2,7,3]) 0  ==>  ((),7)
```

```
let increment = modify (+1) >> get
    ops = replicateM 4 increment
in runState ops 0
  ==> ([1,2,3,4],4)
```

Here's `filter` reimplemented using the `State` monad:

```
rememberElements :: (a -> Bool) -> [a] -> State [a] ()
rememberElements f xs = mapM_ maybePut xs
  where maybePut x = when (f x) (modify (++[x]))

sfilter :: (a -> Bool) -> [a] -> [a]
sfilter f xs = finalState
  where (_, finalState) = runState (rememberElements f xs) []
```

```
sfilter even [1,2,3,4,5]
  ==> [2,4]
```

We can write our own operations that work for all monads. This is made possible by type classes, as we've seen before. If you only use monad operations like `return` and do-notation, the type system will infer a generic type for your function.

```
mywhen b op = if b then op else return ()

mymapM_ op [] = return ()
mymapM_ op (x:xs) = do op x
                       mymapM_ op xs
```

```
*Main> :t mywhen
mywhen :: (Monad m) => Bool -> m () -> m ()
*Main> :t mymapM_
mymapM_ :: (Monad m) => (t -> m a) -> [t] -> m ()
```

We can use these generic operations in each of our example monads:

```
perhapsDecrease :: Int -> Maybe Int
perhapsDecrease x = do
```

```
  mywhen (x<=0) Nothing
  return (x-1)
```

```
perhapsDecrease 2  ==>  Just 1
perhapsDecrease 0  ==>  Nothing
```

```
search :: (Show a, Eq a) => a -> [a] -> Logger ()
search x ys = mymapM_ look ys
  where look y = mywhen (x==y) (msg ("Found "++show y))
```

```
search 3 [1,2,3,4,3,2]  ==>  Logger ["Found 3","Found 3"] ()
```

```
sumPositive :: [Int] -> State Int ()
sumPositive xs = mymapM_ f xs
  where f x = when (x>0) (modify (x+))
```

```
runState (sumPositive [1,-4,2,3]) 0  ==>  ((),6)
```

13.10 Monads are Functors

One useful operation hasn't yet been introduced: `liftM`.

```
liftM :: Monad m => (a->b) -> m a -> m b
liftM f op = do x <- op
                return (f x)
```

The `liftM` operation makes it easy to write code with pure and monadic parts.

```
liftM negate (Just 3)
  ==> Just (-3)

liftM sort $ firsts [[4,6],[2,1,0],[3,3,3]]
  ==> Just [2,3,4]
```

```
runState (liftM negate get) 3
  ==> (-3,3)
```

Does the type of `liftM` look familiar? It's just like the type of `fmap`! In fact, it's easy to define a functor instance for a monad: just set `fmap = liftM`. Since every `Monad` needs to be a `Functor` these days, modern Haskell style prefers `fmap` over `liftM`.

```
fmap :: Functor f => (a->b) -> f a -> f b
```

```
fmap negate (Just 3)
  ==> Just (-3)

fmap sort $ firsts [[4,6],[2,1,0],[3,3,3]]
  ==> Just [2,3,4]

runState (fmap negate get) 3
  ==> (-3,3)
```

13.11 One More Monad

The list monad (that is, the `Monad` instance for `[]`) represents computations with *multiple return values*. It's useful for searching through alternatives. Here's a first example. For every `x` we produce both `x` and `-x`:

```
[1,2,3] >>= \x -> [-x,x]
  ==> [-1,1,-2,2,-3,3]
```

We can filter out unsuitable values by produing an empty list:

```
[1,2,3] >>= \x -> if x>1 then [x] else []
  ==> [2,3]
```

If we're using do-notation the list monad starts to look more like a looping construct:

```
do word <- ["Blue", "Green"]
   number <- [1,2,3]
```

```
    return (word ++ show number)
  ==> ["Blue1","Blue2","Blue3","Green1","Green2","Green3"]
```

More interesting example: find all the pairs in a list that sum to `k`. (The same element twice counts as a pair.)

```
findSum :: [Int] -> Int -> [(Int,Int)]
findSum xs k = do a <- xs
                  b <- xs
                  if (a+b==k) then [(a,b)] else []
```

```
findSum [1,2,3,4,5] 5
  ==> [(1,4),(2,3),(3,2),(4,1)]
```

A final, more complex example. We find all palindromes from a string using the list monad, and then find the longest one.

```
import Data.List (sortBy)

substrings :: String -> [String]
substrings xs = do start <- [0..length xs - 1]
                   end <- [start+1..length xs - 1]
                   return $ drop start $ take end $ xs

palindromesIn :: String -> [String]
palindromesIn xs = do s <- substrings xs
                      if (s==reverse s) then return s else []

longestPalindrome xs = head . sortBy f $ palindromesIn xs
  where f s s' = compare (length s') (length s)  -- longer is
         smaller
```

```
palindromesIn "aabbacddcaca"
  ==> ["a","aa","a","abba","b","bb","b","a","acddca","c","cddc","d","dd
longestPalindrome "aabbacddcaca"
  ==> "acddca"
```

Here's the surprisingly simple implementation of the list monad:

```haskell
instance Monad [] where
  return x = [x]                    -- an operation that produces one
         value
  lis >>= f = concat (map f lis)  -- compute f for all values,
         combine the results
```

We've actually seen the list monad previously in the guise of list comprehensions. Compare this reimplementation of `findSum` to the earlier one that uses `do`-notation.

```haskell
findSum :: [Int] -> Int -> [(Int,Int)]
findSum xs k = [(a,b) | a <- xs, b <- xs, a+b==k ]
```

13.12 Oh Right, IO

As you've probably guessed by now, `IO` is a monad. However the implementations of the `IO` type and `instance Monad IO` are compiler built-ins. You couldn't implement the IO monad just using standard Haskell, unlike `Maybe` monad, `State` monad and other monads we've seen.

However, true side effects fit the monad pattern just like `State` and `Maybe`. Just like with other monads, we're separating the *pure definitions of operations* from the process of *running the operations*. As a bonus, you can use all the generic monad operations (`mapM` and friends) with IO.

Here are some examples of writing IO using monad operations.

```haskell
printTwoThings :: IO ()
printTwoThings = putStrLn "One!" >> print 2

echo :: IO ()
echo = getLine >>= putStrLn

verboseEcho :: IO ()
verboseEcho = getLine >>= \s -> putStrLn ("You wrote: " ++ s)

query :: String -> IO String
query question = putStrLn question >> getLine

confirm :: String -> IO Bool
confirm question = putStrLn question >> fmap interpret getLine
    where interpret "Y" = True
          interpret _ = False
```

```
Prelude> printTwoThings
One!
2

Prelude> verboseEcho
The Iliad
You wrote: The Iliad

Prelude> answer <- query "Why am I here?"
Why am I here?
Good question!
Prelude> answer
"Good question!"

Prelude> b <- confirm "Fire warheads?"
Fire warheads?
no no no no
Prelude> b
False
Prelude> b <- confirm "Make love, not war?"
Make love, not war?
Y
Prelude> b
True
```

13.13 Monads in Other Languages

Once you've gotten familiar with the concept of a monad, you'll start seeing monadlike things in other languages too. The most well-known examples of this are *Option types*, *Java Streams* and *JavaScript promises*. If you know these languages or concepts from before, you might find this section illuminating. If you don't, feel free to skip this.

13.13.1 Optionals

Many langages have an option type. This type is called `Optional<T>` in Java, `std::optional<T>` in C++, `Nullable<T>` in C#, and so on. These types often have behaviour resembling the Haskell `Maybe` monad, for example:

- In Java, `Optional.flatMap` corresponds to `>>=`: it lets you apply a `Function<T,<Optional<U>>` to an `Optional<T>` and get an `Optional<U>`.
- In C#, binary operations get automatically lifted to `Nullable` types. For example, `a + null` becomes `null`.

13.13.2 Streams

Java Streams have a monadlike API too. Streams are about producing many values incrementally. Just like with Optional, the method `Stream.flatMap` lets us take a `Stream<T>`, combine it with a `Function<T,Stream<U>>` and get a `Stream<U>`.

As an example, if `lines` is a `Stream<String>`, `words` takes a `String` and returns a `Stream<String>` and `readInt` takes a `String` and returns an `Integer`, we can write:

```
Stream<Integer> parseNumbers(Stream<String> lines) {
    return lines.flatMap(words).map(read);
}
```

This corresponds to the following Haskell list monad code:

```
parseNumbers :: [String] -> [Int]
parseNumbers strings = fmap read (strings >>= words)
```

```
parseNumbers ["123 456","7 89"]  ==>  [123,456,7,89]
```

13.13.3 Promises

There is much disagreement about whether Promises in JavaScript *really* are monads or not. However, some similarities are obvious.

First, consider the similarities between `Promise.then` and `>>=`. Both take an *operation* (promise or monadic operation), and combine it with a function that returns a new operation.

```
function concatPromises(promise1, promise2) {
  return promise1.then(value1 => promise2.then(value2 =>
         value1+value2));
}
```

```
>> concatPromises(Promise.resolve("abc"),
         Promise.resolve("def")).then(console.log)
abcdef
```

```
concatMonadic :: Monad m => m String -> m String -> m String
concatMonadic op1 op2 = op1 >>= (\value1 -> op2 >>= (\value2 ->
         return (value1++value2)))
```

```
Prelude> concatMonadic (Just "abc") (Just "def")
Just "abcdef"
```

Next, let's consider the similarities between async/await and do-notation. Both are nicer syntaxes for working with the raw `Promise.then` or `>>=` mechanisms. We reimplement `concatPromises` using async/await, and `concatMonadic` using do-notation. Their behaviour stays the same.

```
async function concatPromises(promise1, promise2) {
  let value1 = await promise1;
  let value2 = await promise2;
  return value1+value2;
}
```

```
concatMonadic :: Monad m => m String -> m String -> m String
concatMonadic op1 op2 = do
  value1 <- op1
  value2 <- op2
  return (value1++value2)
```

13.14 Monads: Wrap-up

- The `Monad` type class is a way to represent different ways of *executing recipes*
    - failure (`Maybe`)
    - logging
    - state
    - nondeterminism (the list monad)
    - IO
- You can write monad code in two equivalent ways:
    - Using the `Monad` class operations (`>>=`, `>>`) directly
    - Using `do`-notation
- When `M` is a monad, values of type `M a` are *operations that produce a result of type a*
- Monads are a *design pattern* and a *library* (`mapM` etc)

    - Using common abstractions makes it easier to understand code
    - Reading a `State` operation is easier than deciphering a complicated recursion with state
- Everything you can do with monads, you can also do without them
    - Exception: IO
    - Using a monad often simplifies code
- *Warning*: the internet is full of tutorials trying to explain monads using a simple analogy
    - In my experience, this doesn't work
    - What works is using different monads and slowly getting used to the concept

13.15 Sidenote: Standard Haskell

This and the previous lecture have covered many parts where the GHC version of Haskell differs from standard Haskell 2010. Here's a short list of the changes GHC has made, just so you know:

- `length`, `sum`, `foldr` etc. generalized to work on `Foldable` instead of just lists
- `Functor` and `Applicative` are superclasses of `Monad`
- The `fail` method has been moved from the `Monad` type class to its own `MonadFail` class

13.16 Quiz

What is the expression equivalent to the following do block?

```haskell
do y <- z
   s y
   return (f y)
```

a. `z >> \y -> s y >> return (f y)`
b. `z >>= \y -> s y >> return (f y)`
c. `z >> \y -> s y >>= return (f y)`

What is the type of `\x xs -> return (x : xs)`?

a. `Monad m => a -> [a] -> m [a]`
b. `Monad m => a -> [m a] -> [m a]`
c. `a -> [a] -> Monad [a]`
d. None of the above

What is the type of `\x xs -> return x : xs`?

a. `Monad m => a -> [a] -> m [a]`
b. `Monad m => a -> [m a] -> [m a]`
c. `a -> [a] -> Monad [a]`
d. None of the above

What is the type of `(\x xs -> return x) : xs`?

a. `Monad m => a -> [a] -> m [a]`
b. `Monad m => a -> [m a] -> [m a]`
c. `a -> [a] -> Monad [a]`
d. None of the above

13.17 Exercises

- Set13a
- Set13b

14 Lecture 14: Let’s Use Some Libraries!

Now that you know monads, you pretty much know everything about Haskell to start writing real programs that use libraries to do useful things. This lecture will go over some examples of libraries that are commonly used in such real programs. Using these libraries is also a good way to practice using monads, reading docs, and understanding type errors.

Note! When reading the documentation for the libraries, remember to pay attention to the library version. You can see the versions used on the course in the `tests.cabal` file. The links in the course material always take you to the right version, as does the `stack haddock --open <package>` command. See also Reading Docs in Part 1.

14.1 `Text` and `ByteString`

So far, we’ve been using the Haskell `String` type to work with strings. However `String` is just `[Char]`, a linked list of characters. This is horribly inefficient, both in terms of memory, and in terms of time. Once we move beyond processing short strings and start processing whole files or network requests, a more time efficient string type becomes a must.

There are two types that are used as replacements for `String`, with slightly different semantics:

- `Data.Text` represents *a sequence of Unicode characters*, just like `String`, only more efficient. Used when dealing with text.
- `Data.ByteString` represents *a sequence of bytes*. Used when dealing with binary data.

Additionally, both of these types come in *lazy* and *strict* variants. The docs for `Data.Text` summarize the difference well:

> The strict `Text` type requires that an entire string fit into memory at once. The lazy `Text` type is capable of streaming strings that are larger than memory using a small memory footprint... Each module provides an almost identical API...

All of these types (`Text` and `ByteString`, strict and lazy) offer `pack` and `unpack` functions for converting from and to plain `String`s. The types also come with specialized versions of familiar list functions like `reverse`, `take`, `map` and so on.

14.1.1 Examples with `Text`

Let's go through a short GHCi session demonstrating the use of `Data.Text`. As the documentation says, the `Data.Text` module is designed to be imported *qualified*. We can convert a `String` into a `Text` with the function `T.pack`. Note how a value of type `Text` gets printed just like a `String`.

```
Prelude> import qualified Data.Text as T
Prelude T> :t T.pack
T.pack :: String -> T.Text
Prelude T> phrase = T.pack "brevity is the soul of wit"
Prelude T> :t phrase
phrase :: T.Text
Prelude T> phrase
"brevity is the soul of wit"
```

We can use the functions from `Data.Text` to operate on values of `Text`. Many of these are named like their counterparts for `String`s or lists from `Prelude`.

```
Prelude T> :t T.length
T.length :: T.Text -> Int
Prelude T> T.length phrase
26
Prelude T> T.head phrase
'b'
Prelude T> T.take 4 phrase
"brev"
Prelude T> :t T.words
```

```
T.words :: T.Text -> [T.Text]
Prelude T> T.words phrase
["brevity","is","the","soul","of","wit"]
Prelude T> :t T.map
T.map :: (Char -> Char) -> T.Text -> T.Text
Prelude T> T.map (\c -> if c=='o' then '0' else c) phrase
"brevity is the s0ul 0f wit"
```

A useful detail is that `Text` has a `Monoid` instance that glues `Text` values together. You can also use the functions `T.append` and `T.concat`.

```
Prelude T> phrase <> phrase
"brevity is the soul of witbrevity is the soul of wit"
Prelude T> T.append phrase phrase
"brevity is the soul of witbrevity is the soul of wit"
Prelude T> T.concat [phrase,phrase,phrase]
"brevity is the soul of witbrevity is the soul of witbrevity is the
         soul of wit"
```

If you want to write a recursive function that pattern matches on a `Text` like you would on a `String`, you can use the function `T.uncons :: T.Text -> Maybe (Char, T.Text)` to split a `Text` into a head and a tail. Here's a simple example:

```
countLetter :: Char -> T.Text -> Int
countLetter c t =
  case T.uncons t of
    Nothing -> 0
    Just (x,rest) -> (if x == c then 1 else 0) + countLetter c rest
```

```
Prelude T> countLetter 't' phrase
3
```

14.1.1.1 Strictness and Laziness

Note that `Data.Text` implements the strict `Text` type. You need to use `Data.Text.Lazy` for the lazy variant. As mentioned earlier, one difference between these two types is that the strict type does not work for infinite strings:

```
Prelude T> T.head (T.pack (repeat 'x'))
-- never returns
```

```
Prelude T> import qualified Data.Text.Lazy as TL
Prelude T TL> TL.head (TL.pack (repeat 'x'))
'x'
```

Another practical problem is that you can end up with a mismatch between strict and lazy `Texts` when using libraries. You can usually fix this by using `toStrict` or `fromStrict` as needed.

```
Prelude T TL> lazyPhrase = TL.pack "brevity is the soul of wit"
Prelude T TL> :t lazyPhrase
lazyPhrase :: TL.Text
Prelude T TL> :t phrase
phrase :: T.Text
Prelude T TL> lazyPhrase == phrase

<interactive>: error:
    • Couldn't match expected type ‘TL.Text’
                  with actual type ‘T.Text’
      NB: ‘T.Text’ is defined in ‘Data.Text.Internal’
          ‘TL.Text’ is defined in ‘Data.Text.Internal.Lazy’
    • In the second argument of ‘(==)’, namely ‘phrase’
      In the expression: lazyPhrase == phrase
      In an equation for ‘it’: it = lazyPhrase == phrase

Prelude T TL> :t TL.toStrict
TL.toStrict :: TL.Text -> T.Text
Prelude T TL> :t TL.fromStrict
TL.fromStrict :: T.Text -> TL.Text
Prelude T TL> TL.toStrict lazyPhrase == phrase
True
```

14.1.2 Examples with `ByteString`

We can walk through pretty much the same GHCi session using `ByteString` instead of `Text`. However, note how the `ByteString` is built up from `Word8` values and not `Char` values. A `Char` can represent an arbitrary unicode codepoint like for a character like `'Å'`, but a `Word8` represents a byte: a number from 0 to 255. Unfortunately and somewhat confusingly, `ByteString` values get printed like `Strings`.

```
Prelude> import Data.Word
Prelude Data.Word> import qualified Data.ByteString as B
Prelude Data.Word B> binary = B.pack [99,111,102,102,101,101]
Prelude Data.Word B> :t binary
binary :: B.ByteString
Prelude Data.Word B> :t B.pack
```

```
B.pack :: [Word8] -> B.ByteString
Prelude Data.Word B> binary
"coffee"
Prelude Data.Word B> :t B.length
B.length :: B.ByteString -> Int
Prelude Data.Word B> B.length binary
6
Prelude Data.Word B> :t B.head
B.head :: B.ByteString -> Word8
Prelude Data.Word B> B.head binary
99
Prelude Data.Word B> B.take 4 binary
"coff"
Prelude Data.Word B> :t B.map
B.map :: (Word8 -> Word8) -> B.ByteString -> B.ByteString
Prelude Data.Word B> B.map (+1) binary
"dpggff"
```

The same caveats apply to the differences between strict and lazy `ByteString` as for `Text`:

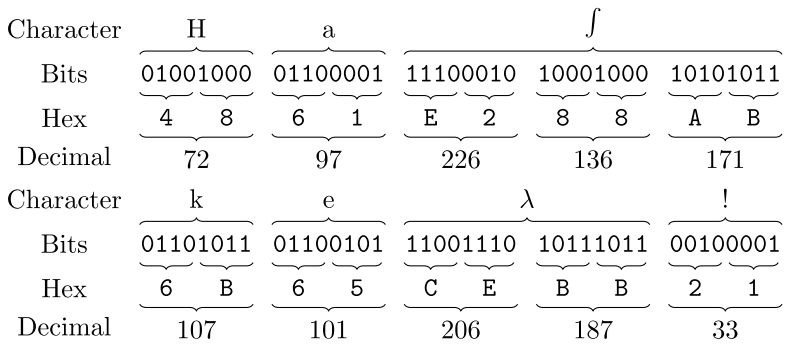
```
Prelude B Data.Char> B.head (B.pack (repeat 99))
-- never returns
Prelude Data.Word B> import qualified Data.ByteString.Lazy as BL
Prelude Data.Word B BL> BL.head (BL.pack (repeat 99))
99
Prelude Data.Word B BL> binary == BL.pack [99]

<interactive>: error:
    • Couldn't match expected type ‘B.ByteString’
                  with actual type ‘BL.ByteString’
      NB: ‘BL.ByteString’ is defined in
          ‘Data.ByteString.Lazy.Internal’
          ‘B.ByteString’ is defined in ‘Data.ByteString.Internal’
    • In the second argument of ‘(==)’, namely ‘BL.pack [99]’
      In the expression: binary == BL.pack [99]
      In an equation for ‘it’: it = binary == BL.pack [99]

Prelude Data.Word B BL> :t BL.toStrict
BL.toStrict :: BL.ByteString -> B.ByteString
Prelude Data.Word B BL> :t BL.fromStrict
BL.fromStrict :: B.ByteString -> BL.ByteString
Prelude Data.Word B BL> binary == BL.toStrict (BL.pack [99])
False
```

14.1.3 Sidenote: Encodings

You might be wondering why on earth do we have both Text and ByteString. The difference is subtle but real. When we operate on `Text` we operate character by character, regardless of what those characters are and how they are encoded. When we operate on `ByteString` we operate on bytes, regardless of what those bytes represent.

Characters, numbers, and data structures are abstractions that help us humans to deal with complex programming tasks. The computer memory is essentially just an enormous sequence of bytes. The machine doesn't care how we interpret those bytes. The essential difference between `Text` and `ByteString` is the way how the bytes are grouped and interpreted.

To illustrate this difference, we'll look at the UTF-8 text encoding. A text encoding is a method for representing *characters* as *bytes*. UTF-8 can represent all the millions of characters defined by Unicode. Because bytes can only store values from 0 to 255, this means that a character can get encoded into multiple bytes. The bits and bytes of the UTF-8 string "Ha∫keλ!" can be interpreted in various ways:



(If you see different characters in the picture and in "Ha∫keλ!", it means that your browser is either interpreting the encoding incorrectly or the font you're using doesn't support all of the characters.)

If we read the same stream of bits using a different encoding, we'll see other characters. For example the string above would be interpreted as "Haâˆ«keÎ»!" using the Latin-1 text encoding.

By the way, when dealing with raw binary data, it's often convenient to use the hexadecimal number system which uses a single symbol `0`, `1`, ..., `9`, `A`, `B`, ..., `F` to represent all of the sixteen possible combinations of four bits. We don't need hexadecimal in this course but you can check out Wikipedia if you're interested in learning more about hexadecimal.

We can explore the same example using code. The function `Data.Text.Encoding.encodeUtf8 :: Text -> ByteString` encodes the characters in a Text into bytes in a ByteString using UTF-8.

```
Prelude> import qualified Data.Text as T
Prelude T> import qualified Data.ByteString as B
```

```
Prelude T B> T.length (T.pack "haskell")
7
Prelude T B> T.length (T.pack "Haʃkeλ!")
7
Prelude T B> import Data.Text.Encoding
Prelude T B Data.Text.Encoding> encodeUtf8 (T.pack "haskell")
"haskell"
Prelude T B Data.Text.Encoding> encodeUtf8 (T.pack "Haʃkeλ!")
"Ha\226\136\171ke\206\187!"
Prelude T B Data.Text.Encoding> B.length (encodeUtf8 (T.pack
         "haskell"))
7
Prelude T B Data.Text.Encoding> B.length (encodeUtf8 (T.pack
         "Haʃkeλ!"))
10
```

If we are processing ASCII text, that is, characters that can be represented using single bytes, we can use `Text` and `ByteString` interchangeably. There are namespaces `Data.ByteString.Char8` and `Data.ByteString.Lazy.Char8` that offer functions that operate on `ByteStrings` using `Char` values instead of `Word8`. However, care must be taken to make sure all the characters really are plain ASCII characters, or surprising things will happen.

```
Prelude T B> import qualified Data.ByteString.Char8 as B8
Prelude T B B8> B8.pack "abc"
"abc"
Prelude T B B8> :t B8.pack
B8.pack :: String -> B.ByteString
Prelude T B B8> :t B.pack
B.pack :: [Word8] -> B.ByteString
Prelude T B B8> B8.cons 'a' (B8.pack "bc")
"abc"
Prelude T B B8> putStrLn (B8.unpack (B8.pack "€λ훈"))  -- non-ASCII
         characters get truncated
¬»È
Prelude T B B8> putStrLn (T.unpack (T.pack "€λ훈"))
€λ훈
```

14.2 Monads: Recap

The next libraries we'll look at work inside the IO monad. Here's a short recap of what we learned about monads in the last lecture.

- When `M` is a monad, values of type `M X` are *operations* that can be *executed* to *produce* a value of type `X`.
- Monadic operations can be implemented using

    - the methods of the `Monad` type class (`return`, `>>=`, `>>`),
    - `do`-notation,
    - and library functions like `mapM`.
- Unlike in other languages, `return` is not a keyword and does not cause an operation to stop executing. Instead, `return x` *is the operation that always produces x and does nothing else*.
- This is what `do`-notation looks like:

```
foo y = do
  operation1          -- run an operation
  val <- operation2   -- run an operation and keep the produced value
  operation3 val y    -- run an operation with parameters
  mapM_ (\x -> operation4 val x) things  -- use a generic monad
         operation and a lambda
  operation5 val      -- the final line of the do decides which value
         the whole block produces
```

14.3 Writing a HTTP Server: WAI and Warp

Sometimes it feels like everything in the world happens over HTTP and Web Apis. Your web browser, your smartphone apps, your bank, your coffee pot, and even your doorbell, all talk to servers using the HTTP protocol.

Let's look at how we can set up a simple HTTP server in Haskell. The standard low level components for this are called WAI and Warp. WAI, the Web Application Interface gives us a way to define how HTTP requests are handled. Warp is a simple HTTP server that runs the logic we have defined using WAI. That probably sounds a bit abstract right now, but a simple example will help.

The file `exercises/Examples/HelloServer.hs` implements a HTTP server that always responds with "Hello World!". You can try it out by going to the `exercises/Examples` directory and running it with `stack runhaskell HelloServer.hs`. After that you can visit http://localhost:3421 in your browser to see the response from the server.

```
module Examples.HelloServer where

import qualified Data.ByteString.Lazy.Char8 as BL
import Network.HTTP.Types.Status (status200)
import Network.Wai (Application, responseLBS)
import Network.Wai.Handler.Warp (run)

port :: Int
port = 3421
```

```
main :: IO ()
main = run port application

-- type Application = Request -> (Response -> IO ResponseReceived) -
         > IO ResponseReceived
application :: Application
application request respond =
  respond (responseLBS status200 [] (BL.pack "Hello World!"))
```

Let's look at the types in this example. There's a lot going on here. First off, `Application` is a *type alias* for a thing that implements the logic of a web server. The `run` function from Warp can run `Applications`:

```
run :: Port -> Application -> IO ()
type Application = Request -> (Response -> IO ResponseReceived) ->
         IO ResponseReceived
```

We'll talk about what `Request` and `Response` are soon, but from this type we can see that an `Application` is an IO operation that takes as arguments a request of type `Request`, and an IO operation `respond :: Response -> IO ResponseReceived`. Arguments like `respond` are called *callbacks* in many contexts. They allow us to call back to the library who called the application. The `Application` operation has to produce the same special `ResponseReceived` type that `respond`. You can think of this type as a token proving that `respond` was called by the `Application`.

That might sound intimidating: but looking at the code things are relatively simple: our `server` is an `Application` and takes two parameters: `request` and `respond`.

WAI uses lots of types like `Port`, `Request`, `Response`, `Status` to represent HTTP concepts. It's useful to look these up in the documentation when you bump into them. For example `Port` is just an alias for `Int`. As another example, we can see the `responseLBS` function has the type

```
responseLBS :: Status -> ResponseHeaders -> ByteString -> Response
```

where `Status` is defined in `Network.HTTP.Types.Status`, `ResponseHeaders` is a type alias for `[Header]` from `Network.HTTP.Types.Header`, `ByteString` is a lazy `ByteString`, and the result type `Response` is defined by `Network.WAI`.

Finally, note that we take a shortcut and use the function `Data.ByteString.Lazy.Char8.pack` to convert a `String` into a `ByteString`. This only works for ASCII text.

A web server that always responds with the same text isn't that interesting. Let's look at how we can give different responses to different requests next. There are many parts in a HTTP request, but for this lecture we're going to focus on the *path*. In a URL like `http://example.com/abcd/ef/file`, the `/abcd/ef/file` part is the path. WAI has the function

```
pathInfo :: Request -> [Text]
```

This gives us the path of the requested URL, split at the `/` characters.

The file `exercises/Examples/PathServer.hs` implements a web server that has three different pages:

- http://localhost:3421/source is the source of the application itself, read from the file system
- http://localhost:3421/secret/file is a secret string
- http://localhost:3421/anything/else - for all other paths, a "Not found: anything/else" text is shown

As before, you can run the server by going into the `exercises/Examples` directory, and running `stack runhaskell PathServer.hs`.

14.4 Working With a Database: sqlite-simple

After implementing a HTTP server we can participate in the global graph of applications talking to each other that's called the internet. But what use is talking if we can't remember? A real application needs to be able to *persist data* even when it is restarted. A common way to accomplish this is to use a database.

There are many different kinds of databases, but arguably the most widely used simple database is SQLite. SQLite is a library that lets you store data in a file and process it using SQL, the Structured Query Language. With SQLite there is no need to run a separate database server a la PostgreSQL or MySQL.

In case you're not familiar with SQL, don't worry, you won't need to write any queries of your own in the exercises. If you'd like to learn a bit of SQL now, there are lots of tutorials on the web. See W3Schools, SQL Zoo or Codecademy.

There are many libraries for Haskell for using SQLite, but we'll look at one called sqlite-simple here. Let's explore the library in GHCi for a bit.

All the functions live inside `Database.SQLite.Simple`. You can open a database by giving a filename to `open`, which is an IO operation that produces a `Connection`.

```
Prelude> import Database.SQLite.Simple
Prelude Database.SQLite.Simple> :t open
open :: String -> IO Connection
Prelude Database.SQLite.Simple> db <- open "example.sqlite"
```

To run an SQL query you can use IO operation `query_`, which takes a `Connection` and a `Query`, and produces a list of results. The `Query` type is just a simple `newtype` around `Text`. The result type of `query_`is polymorphic: you can read any type that satisfies the `FromRow` type class from the database. If this feels confusing, compare it to the type of `read`: `Read a => String -> a`. The `FromRow` class is like `Read` for this database: it represents types that can be read out of the database. Anyway, let's read the number `1` from the database:

```
Prelude Database.SQLite.Simple> :t query_
query_ :: FromRow r => Connection -> Query -> IO [r]
Prelude Database.SQLite.Simple> :info Query
newtype Query = Query {fromQuery :: Data.Text.Internal.Text}
    -- Defined in ‘Database.SQLite.Simple.Types’
    -- ... rest of output omitted
Prelude Database.SQLite.Simple> import qualified Data.Text as T
Prelude Database.SQLite.Simple T> q = Query (T.pack "SELECT 1;")
Prelude Database.SQLite.Simple T> res <- query_ db q :: IO [[Int]]
Prelude Database.SQLite.Simple T> res
[[1]]
```

By the way, all of these initial examples use simple `SELECT x, y, z;` queries that just return constant data. We'll worry about actual tables in the database later.

Without the type signature, we get an error from GHCi, which can't decide which type we want to read out from the database:

```
Prelude Database.SQLite.Simple T> res <- query_ db q

<interactive>:17:8: error:
    • Ambiguous type variable ‘r0’ arising from a use of ‘query_’
      prevents the constraint ‘(FromRow r0)’ from being solved.
      Probable fix: use a type annotation to specify what ‘r0’
        should be.
      -- rest of error omitted
```

Before we go on, let's have a closer look at `FromRow`. If you've bumped into SQL before, you know that an SQL query returns a number of *rows*, and each row consists of a number of *values* (also called *columns*). To be able to interpret the

result of an SQL query into Haskell data, we need a way to interpret these values and rows. Thus sqlite-simple defines two classes, `FromField` and `FromRow`, and a bunch of instances like the following. (You can find these instances from the docs or by asking GHCi with `:info FromRow` etc.)

```
instance FromField Int
instance FromField Bool
instance FromField String
instance FromField Text
instance FromField a => FromRow [a]
instance (FromField a, FromField b) => FromRow (a,b)
instance (FromField a, FromField b, FromField c) => FromRow (a,b,c)
```

In essence, basic Haskell datatypes fulfill the `FromField` class, and various Haskell collections fulfill the `FromRow` class. Our earlier example was using the `FromRow [a]` and `FromField Int` instances to get a `[[Int]]` out of `query_`. Here's a simple query that uses some other datatypes:

```
Prelude Database.SQLite.Simple T> q = Query (T.pack "SELECT 1, true,
         'string';")
Prelude Database.SQLite.Simple T> query_ db q :: IO
         [(Int,Bool,String)]
[(1,True,"string")]
```

What happens if the SQL and Haskell types don't match? Well, you get a runtime error, just like if you try to invoke `read "True" :: Int`.

```
Prelude Database.SQLite.Simple T> query_ db q :: IO [(Int,Int,Int)]
*** Exception: ConversionFailed {errSQLType = "TEXT", errHaskellType
         = "Int", errMessage = "need an int"}
```

To mirror the `FromRow` and `FromField` classes, sqlite-simple also defines the `ToRow` and `ToField` classes for writing into the database. Here's the type of the `query` function, which allows us to use *parameterized queries*.

```
query :: (ToRow q, FromRow r) => Connection -> Query -> q -> IO [r]
```

And here are some instances for `ToRow` and `ToField`:

```
instance ToField Int
instance ToField Bool
instance ToField String
instance ToField Text
instance ToField Int

instance ToField a => ToRow [a]
instance (ToField a, ToField b) => ToRow (a, b)
instance (ToField a, ToField b, ToField c) => ToRow (a, b, c)
instance ToField a => ToRow (Only a)
```

Parameterized queries use a `?` character to denote slots where parameters can be passed in. Here's a simple example:

```
Prelude Database.SQLite.Simple T> input = (1,"hello") ::
         (Int,String)
Prelude Database.SQLite.Simple T> parameterized = Query (T.pack
         "SELECT ?+1, true, ?;")
Prelude Database.SQLite.Simple T> query db parameterized input :: IO
         [(Int,Bool,String)]
[(2,True,"hello")]
```

Note! When performing a query with only one parameter, there are two `ToRow` instances you can use: `ToField a => ToRow [a]` and `ToField a => Only a`. The `Only` datatype is defined in `Data.Tuple.Only` and is kind of a workaround for the fact that Haskell doesn't have one-element tuples. Alternatively, a list of size 1 works as well. The same applies for queries that return rows with only one column: you can use either `[[X]]` or `[Only X]` as the return type. Here's an example:

```
Prelude Database.SQLite.Simple T> q = Query (T.pack "SELECT
         lower(?);")
Prelude Database.SQLite.Simple T> query db q (Only "HELLO") :: IO
         [Only String]
[Only {fromOnly = "hello"}]
Prelude Database.SQLite.Simple T> query db q ["HELLO"] :: IO
         [[String]]
[["hello"]]
```

That's pretty much all you need to know about sqlite-simple: `open`, `query_`, `query`, `FromRow`, `ToRow`. Oh right, one more thing. There are the functions `execute` and `execute_` if you don't need the result of the query. They're useful for inserting things into the database, for example.

```
execute_ :: Connection -> Query -> IO ()
execute :: ToRow q => Connection -> Query -> q -> IO ()
```

You'll find an example program that uses sqlite-simple to maintain a phone book under `exercises/Examples/Phonebook.hs`. The program keeps the phonebook in a file called `phonebook.db` and works like this (run from the `exercises/Examples` directory in the course repository):

```
$ stack runhaskell Phonebook.hs
(a)dd or (q)uery?
a
Name?
bob
Phone?
1234
$ stack runhaskell Phonebook.hs
(a)dd or (q)uery?
a
Name?
bob
Phone?
5678
$ stack runhaskell Phonebook.hs
(a)dd or (q)uery?
a
Name?
samantha
Phone?
1357
$ stack runhaskell Phonebook.hs
(a)dd or (q)uery?
q
Name?
bob
2 numbers:
["1234"]
["5678"]
```

PS. If you're devastated by the lack of compile-time type checking for SQL queries, you can have a look at some of the more advanced SQL libraries for Haskell like Beam or Opaleye. This course is using sqlite-simple for simplicity and to avoid dwelling on the details of SQL too much.

14.5 Exercises

- Set14a: Text & ByteString

- Set14b: HTTP & SQLite

15 Lecture 15: You're Valid Even Without Monads

15.1 Introduction to Applicatives

The `Applicative` type class is a middle ground between `Functor` (which you can't do that much with) and `Monad` (which pretty much allows you to write arbitrary programs). Reasons to use `Applicative` instead of `Monad` include:

- Performance: since `Applicative` allows less operations, it can be optimized better than `Monad`.
- Simplicity: an `Applicative` interface is easier to reason about.
- Necessity: there just is no way to define a `Monad` instance for your type, but there is an `Applicative` instance. This is rare.

So what is an `Applicative`? Let's look at a definition.

```
class Functor f => Applicative f where
  pure :: a -> f a
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  -- other operations omitted for now
```

So an `Applicative` is a `Functor` that allows us to build singleton values via `pure`, and combine two values into one using `liftA2`. This adds a lot of power compared to a bare functor. Computations using functors are necessarily linear:
`fmap :: (a -> b) -> f a -> f b` takes in one functorial value, and outputs another. By contrast, `pure` takes in no functorial value and outputs one, and `liftA2` takes in two and returns one.

Sidenote: the term Applicative comes from the term Applicative Functor, which sounds like it comes from Category Theory, but was actually introduced in a programming paper.

That's enough abstract mumbo jumbo for now. Let's look at what sort of computations we can express using Applicative operations (and `fmap`). We'll start with the `Maybe` applicative. Here's a streamlined definition:

```
instance Applicative Maybe where
  pure x = Just x
```

```
  liftA2 f (Just x) (Just y) = Just (f x y)
  liftA2 f _        _        = Nothing
```

You'll see the definition uses the same type of failure propagation as the `Monad Maybe` instance. Let's use this when parsing monetary values:

```
data Currency = EUR | USD
  deriving (Show, Eq)
data Money = Money Int Currency
  deriving (Show, Eq)

parseCurrency :: String -> Maybe Currency
parseCurrency "e" = pure EUR
parseCurrency "€" = pure EUR
parseCurrency "$" = pure USD
parseCurrency _ = Nothing

parseAmount :: String -> Maybe Int
parseAmount = readMaybe

parseMoney :: String -> String -> Maybe Money
parseMoney amountString currencyString =
  liftA2 Money (parseAmount amountString) (parseCurrency
         currencyString)



parseMoney "123" "€"  ==> Just (Money 123 EUR)
parseMoney "45" "$"   ==> Just (Money 45 USD)
parseMoney "4x" "€"   ==> Nothing
parseMoney "45" "£"   ==> Nothing
```

That worked out nicely. However, if we try to expand on this we soon run into the limits of `Applicative`. For example, consider this `sumMoney` function that sums `Money` values but fails if they are not in the same currency:

```
sumMoney :: Money -> Money -> Maybe Money
sumMoney (Money a c) (Money b c')
    | c == c'   = Just (Money (a+b) c)
    | otherwise = Nothing
```

We can't apply it to two `Maybe Money` values using `Applicative` operations. We need the `Maybe` monad for that:

```haskell
example :: Maybe Money
example = do x <- parseMoney "123" "€"
             y <- parseMoney "45" "$"
             sumMoney x y
```

If we try to use `liftA2`, we're stuck with a `Maybe (Maybe Money)` type. In addition, we can now get two different types of failures: `Nothing` and `Just Nothing`, depending on what level the error happens on. This would be a clear case for switching to a `Monad` instance.

```
liftA2 sumMoney (parseMoney "123" "e") (parseMoney "45" "€")
  ==> Just (Just (Money 168 EUR))
liftA2 sumMoney (parseMoney "123" "e") (parseMoney "45" "$")
  ==> Just Nothing
liftA2 sumMoney (parseMoney "123" "e") (parseMoney "xxx" "e")
  ==> Nothing
```

Sidenote: the name `liftA2` sounds a bit cumbersome, but it's an analogy with the `liftM`, `liftM2` and so on functions for monads. Recall that `liftM` was just `fmap`, so perhaps `liftA2` should've been called `fmap2`.

15.2 The List Applicative

Let's have a look at the applicative instances for another `Functor` we've seen. The `Applicative` instance for the list functor goes through all possible combinations of values (just like the list monad). Here's the instance:

```haskell
instance Applicative [] where
  pure x = [x]
  liftA2 f xs ys = [f x y | x <- xs, y <- ys]
```

And here's an example: generating some phrases.

```haskell
things :: [String]
things = ["tangerine","bandit","diamond"]

fruits :: [String]
fruits = ["apple", "tangerine"]

phrases :: [String]
phrases = liftA2 combine things fruits
  where combine t f = "a " ++ t ++ " the size of a " ++ f
```

```
bunches = liftA2 copy [1,2,3] fruits
  where copy n f = unwords (replicate n f)
```

```
phrases ==> ["a tangerine the size of a apple",
             "a tangerine the size of a tangerine",
             "a bandit the size of a apple",
             "a bandit the size of a tangerine",
             "a diamond the size of a apple",
             "a diamond the size of a tangerine"]

bunches ==> ["apple","tangerine",
             "apple apple","tangerine tangerine",
             "apple apple apple","tangerine tangerine tangerine"]
```

15.3 New Operators

There are a handful of operators for Applicatives that are quite handy. They are <$>, <*>, <* and *>.

Let's start with <$>, which is just an infix version of `fmap`:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

```
not <$> Just True   ==> Just False
not <$> Nothing     ==> Nothing
negate <$> [1,2,3]  ==> [-1,-2,-3]
```

That's kinda nice on its own, but it really gets to shine when combined with this Applicative operator:

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

The type tells you what <*> does: it's function application *lifted* to an applicative. Here are some standalone examples:

```
Just not <*> Just True    ==> Just False
Nothing  <*> Just True    ==> Nothing
```

```
Just not <*> Nothing      ==> Nothing
[(+1),(*2)] <*> [10,100]  ==> [11,101,20,200]
```

The real magic happens when we combine <$> and <*>: we can then lift functions of arbitrarily many arguments to an Applicative!

```
say :: String -> Int -> String -> String
say x i y = x ++ " has " ++ show i ++ " " ++ y
```

```
say <$> Just "haskell" <*> Just 99 <*> Just "operators"
  ==> Just "haskell has 99 operators"
say <$> Nothing <*> Just 99 <*> Just "operators"
  ==> Nothing
say <$> ["bob","jake"] <*> [2,3] <*> ["bananas","cars"]
  ==> ["bob has 2 bananas",
       "bob has 2 cars",
       "bob has 3 bananas",
       "bob has 3 cars",
       "jake has 2 bananas",
       "jake has 2 cars",
       "jake has 3 bananas",
       "jake has 3 cars"]
```

What's going on here? Let's step through the evaluation. The key is that each <*> partially applies one more argument to the function.

```
    say <$> Just "haskell" <*> Just 99 <*> Just "operators"
=== ((say <$> Just "haskell") <*> Just 99) <*> Just "operators"
=== (fmap say (Just "haskell") <*> Just 99) <*> Just "operators"
==> (Just (say "haskell") <*> Just 99) <*> Just "operators"
==> Just (say "haskell" 99) <*> Just "operators"
==> Just (say "haskell" 99 "operators")
==> Just "haskell has 99 operators"
```

Perhaps looking at the types will make it clearer:

```
say <$> Just "haskell"                                  :: Maybe
         (Int -> String -> String)
say <$> Just "haskell" <*> Just 99                      :: Maybe (
         String -> String)
```

```
say <$> Just "haskell" <*> Just 99 <*> Just "operators" :: Maybe (
         String)
```

The next two operators are a bit simpler:

```
(*>) :: Applicative f => f a -> f b -> f b
x *> y = liftA2 (\a b -> b) x y

(<*) :: Applicative f => f a -> f b -> f a
x <* y = liftA2 (\a b -> a) x y
```

You can compare the types to a more familiar operator:

```
(>>) :: Monad m => m a -> m b -> m b
```

What the operators <* and *> mean is: run both of these operations, but only keep one result. The arrow points to the result that' s kept:

```
Just 1 *> Just 2  ==> Just 2
Just 1 <* Just 2  ==> Just 1
Just 1 <* Nothing ==> Nothing
Nothing <* Just 2 ==> Nothing
```

These operators might seem trivial, but they' re useful when combining checks. For example:

```
decrease :: Int -> Maybe Int
decrease i = if i>0 then Just (i-1) else Nothing

small :: Int -> Maybe Int
small i = if i<10 then Just i else Nothing

decreaseSmall :: Int -> Maybe Int
-- do what decrease does, but fail if small fails
decreaseSmall i = decrease i <* small i
```

```
decreaseSmall 4   ==> Just 3
decreaseSmall 0   ==> Nothing
decreaseSmall 11  ==> Nothing
```

Now that we've seen all these operators, we can understand the full definition of `Applicative`. All the operators have definitions in terms of `liftA2`, so it's enought to define `liftA2` and `pure` when implementing an `Applicative` instance.

```
class Functor f => Applicative f where
  pure :: a -> f a
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (<*>) :: f (a -> b) -> f a -> f b
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
```

15.4 The Validation Applicative

Let's look at an Applicative that's a bit more interesting than Maybe or lists. Often in programming we need to *validate* some inputs from the user. In these cases it's useful to gather together all the errors that the input might have. The file `exercises/Examples/Validation.hs` implements the `Validation` datatype:

```
data Validation a = Ok a | Errors [String]
  deriving (Show,Eq)
```

The `Applicative` instance for `Validation` works like this:

```
liftA2 (+) (Ok 1) (Ok 2)
  ==> Ok 3
liftA2 (+) (Errors ["oh no"]) (Errors ["boom"])
  ==> Errors ["oh no","boom"]
```

Note how in contrast to the `Maybe` Applicative, we have many different kinds of failures.

Here's a worked example that introduces some helpers and then uses them to congratulate somebody on their birthday:

```
invalid :: String -> Validation a
invalid err = Errors [err]

check :: Bool -> String -> a -> Validation a
check b err x
  | b = pure x
  | otherwise = invalid err
```

```
birthday :: String -> Int -> Validation String
birthday name age = liftA2 congratulate checkedName checkedAge
  where checkedName = check (length name < 10) "Name too long" name
        checkedAge = check (age < 99) "Too old" age
        congratulate n a = "Happy "++show a++"th birthday "++n++"!"
```

```
birthday "Guy" 31
  ==> Ok "Happy 31th birthday Guy!"
birthday "Guybrush Threepwood" 31
  ==> Errors ["Name too long"]
birthday "Yog-sothoth" 10000
  ==> Errors ["Name too long","Too old"]
```

Oh right, here are the `Functor` and `Applicative` instances for `Validation`:

```
instance Functor Validation where
  fmap f (Ok x) = Ok (f x)
  fmap _ (Errors e) = Errors e

instance Applicative Validation where
  pure x = Ok x
  liftA2 f (Ok x)      (Ok y)      = Ok (f x y)
  liftA2 f (Errors e1) (Ok y)      = Errors e1
  liftA2 f (Ok x)      (Errors e2) = Errors e2
  liftA2 f (Errors e1) (Errors e2) = Errors (e1++e2)
```

The definition of `liftA2` for `Validation` shows that the errors are collected together left-to-right. This can be seen in the example above where the expression `liftA2 congratulate checkedName checkedAge` outputs the error (`"Name too long"`) from `checkedName` first, and the error (`"Too old"`) from `checkedAge` last.

15.5 Validating Lists: `traverse`

So far we've dealt with fixed-size things and Applicatives: we've applied a function of two or three arguments to some things. What if we have an arbitrary amount of inputs? What if we need to validate a list?

Let's look at some ways to implement a function like this:

```
allPositive [1,2,3]
  ==> Ok [1,2,3]
allPositive [1,2,3,-4]
```

```
  ==> Errors ["Not positive: -4"]
allPositive [1,-2,3,-4]
  ==> Errors ["Not positive: -2","Not positive: -4"]
```

As always, when working with lists, pattern matching and recursion are usually the way to go. Here's a recursive solution:

```
allPositive :: [Int] -> Validation [Int]
allPositive [] = Ok []
allPositive (x:xs) = liftA2 (:) checkThis checkRest
  where checkThis = check (x>=0) ("Not positive: "++show x) x
        checkRest = allPositive xs
```

It's a bit of a chore to always spell out a recursion like this. If we were working in a `Monad` we could just use a helper like `mapM`:

```
mapM (\x -> if x>=0 then Just x else Nothing) [1,2,3]
  ==> Just [1,2,3]
mapM (\x -> if x>=0 then Just x else Nothing) [1,2,3,-4]
  ==> Nothing
```

The equivalent of `mapM` for `Applicative` is called `traverse`. It's a member of the type class `Traversable`:

```
traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f
         (t b)
```

That's one heck of a type signature, so let's simplify it a bit. Lists are `Traversable`, so we can specialize this type into:

```
traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
```

That looks like what we need! For Applicatives that are also Monads, `traverse` is just another name for `mapM`:

```
traverse (\x -> if x>=0 then Just x else Nothing) [1,2,3]
  ==> Just [1,2,3]
traverse (\x -> if x>=0 then Just x else Nothing) [1,2,3,-4]
  ==> Nothing
```

But for our `Validation`, which isn't a `Monad`, `traverse` is exactly what we want:

```
allPositive :: [Int] -> Validation [Int]
allPositive xs = traverse checkNumber xs
  where checkNumber x = check (x>=0) ("Not positive: "++show x) x
```

```
allPositive [1,2,3]
  ==> Ok [1,2,3]
allPositive [1,2,3,-4]
  ==> Errors ["Not positive: -4"]
allPositive [1,-2,3,-4]
  ==> Errors ["Not positive: -2","Not positive: -4"]
```

Note how `traverse` for `Validation` collects all the errors together, in the order they occur in the original list.

PS. In fact, `Validation` is one of the few examples of an `Applicative` that can't be a `Monad`. Can you figure out why?

15.6 Sidenote: `Traversable`

So what things are `Traversable`? Many familiar structures. Here are some examples:

```
decrease :: Int -> Maybe Int
decrease i = if i>0 then Just (i-1) else Nothing
```

```
-- Lists are Traversable
traverse decrease [1,2,3] ==> Just [0,1,2]
traverse decrease [1,0,3] ==> Nothing

-- Arrays are Traversable
traverse decrease (array (1,3) [(1,10),(2,11),(3,12)])
         ==> Just (array (1,3) [(1,9),(2,10),(3,11)])

-- Maps are Traversable
traverse decrease (M.fromList [("a",1),("b",2)])
         ==> Just (M.fromList [("a",0),("b",1)])
traverse decrease (M.fromList [("a",1),("b",0)])
         ==> Nothing

-- Either is Traversable
traverse decrease (Left "abc") ==> Just (Left "abc")
```

```
traverse decrease (Right 3)    ==> Just (Right 2)
traverse decrease (Right 0)    ==> Nothing
```

So `Traversable` is a type class for all sorts of containers, kind of like `Foldable`. Indeed, if you look at the definition, `Traversable` is a subclass of `Foldable`. It also turns out that `traverse` and `mapM` are methods of the class!

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
```

It can be hard to keep the types straight here. Let's go back to the type of `traverse`:

```
traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f
         (t b)
```

Here we have two functors: `t` and `f`. The `t` functor is also a `Foldable` and a `Traversable` and the `f` functor is also an `Applicative`. The `traverse` function lets us run `f`-operations inside a `t`-container.

Don't worry if that feels abstract. In practice, you pretty much always use `traverse` on lists.

15.7 Dealing with Failure: `Alternative`

If you play around with applicatives a bit you'll start to notice some limits to their power. For example, when writing parsers like we did in the `parseMoney` example, it would be nice to be able to try a couple of different parsers and take any non-failure result. This is easy enough to write for a concrete Applicative like `Maybe`, as can be seen below.

```
data Answer = Yes | No
  deriving (Show, Eq)

parseYes :: String -> Maybe Answer
parseYes "y" = Just Yes
parseYes "yes" = Just Yes
parseYes "maybe" = Just Yes
parseYes _ = Nothing

parseNo :: String -> Maybe Answer
parseNo "n" = Just No
```

```
parseNo "no" = Just No
parseNo "maybe" = Just No
parseNo _ = Nothing

eitherOf :: Maybe x -> Maybe x -> Maybe x
eitherOf (Just x) _  = Just x
eitherOf Nothing  mx = mx

parseAnswer :: String -> Maybe Answer
-- prefer positive answers!
parseAnswer s = eitherOf (parseYes s) (parseNo s)
```

```
parseAnswer "yes"    ==> Just Yes
parseAnswer "y"      ==> Just Yes
parseAnswer "n"      ==> Just No
parseAnswer "maybe"  ==> Just Yes
parseAnswer "x"      ==> Nothing
```

How could we generalize `eitherOf`? We can't give it the type `Applicative f => f x -> f x -> f x` because then the implementation would need to be effectively something like `eitherOf a b = liftA2 something a b`, but then `eitherOf Nothing (Just x)` would be `Nothing` (since that's how the Applicative instance works)!

It turns out we need a new type class: `Alternative`. An Alternative adds two operations to Applicative: `empty` means no results, and `<|>` means combining results.

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  -- some other operations omitted
```

Now we can rewrite our parsing code using general operations:

```
data Answer = Yes | No
  deriving (Show, Eq)

parseYes :: Alternative f => String -> f Answer
parseYes "y" = pure Yes
parseYes "yes" = pure Yes
parseYes "maybe" = pure Yes
parseYes _ = empty

parseNo :: Alternative f => String -> f Answer
parseNo "n" = pure No
```

```
parseNo "no" = pure No
parseNo "maybe" = pure No
parseNo _ = empty

parseAnswer :: Alternative f => String -> f Answer
parseAnswer s = parseYes s <|> parseNo s
```

We can also pick which `Alternative` we run our parser in to get different behaviours. `Maybe` gives us only one result, while `[]` gives us all possible results.

```
> parseAnswer "yes" :: Maybe Answer
Just Yes
> parseAnswer "maybe" :: Maybe Answer
Just Yes
> parseAnswer "yes" :: [Answer]
[Yes]
> parseAnswer "maybe" :: [Answer]
[Yes,No]
```

The `Alternative` instances for `[]` and `Maybe` are unsurprising:

```
instance Alternative [] where
  empty = []
  (<|>) = (++)

instance Alternative Maybe where
  empty = Nothing
  Just x  <|> _  = Just x
  Nothing <|> mx = mx
```

The `Validation` type is also an `Alternative`. The instance collects together all error messages, just like the `Applicative` instance did.

```
instance Alternative Validation where
  empty = Errors []
  Ok x <|> _ = Ok x
  Errors e1 <|> Ok y = Ok y
  Errors e1 <|> Errors e2 = Errors (e1++e2)
```

Here's a final example: validating contact information, which is either a phone number or an email address.

```haskell
data ContactInfo = Email String | Phone String
  deriving Show

validateEmail :: String -> Validation ContactInfo
validateEmail s = check (elem '@' s) "Not an email: should contain a
         @" (Email s)

checkLength :: String -> Validation ContactInfo
checkLength s = check (length s <= 10) "Not a phone number: should
         be at most 10 digits" (Phone s)

checkDigits :: String -> Validation ContactInfo
checkDigits s = check (all isDigit s) "Not a phone number: should be
         all numbers" (Phone s)

validatePhone :: String -> Validation ContactInfo
validatePhone s = checkDigits s *> checkLength s

validateContactInfo :: String -> Validation ContactInfo
validateContactInfo s = validateEmail s <|> validatePhone s
```

```haskell
validateContactInfo "user@example.com"
  ==> Ok (Email "user@example.com")
validateContactInfo "01234"
  ==> Ok (Phone "01234")
validateContactInfo "01234567890"
  ==> Errors ["Not an email: should contain a @","Not a phone
         number: should be at most 10 digits"]
validateContactInfo "01234567890x"
  ==> Errors ["Not an email: should contain a @",
              "Not a phone number: should be all numbers",
              "Not a phone number: should be at most 10 digits"]
validateContactInfo "x"
  ==> Errors ["Not an email: should contain a @",
              "Not a phone number: should be all numbers"]
```

Note how here, just like in previous examples, the errors are collected left-to-right: the errors from `validateEmail` come before the errors from `validatePhone`. The error from `checkDigits` comes before the error from `checkLength`.

15.8 Sidenote: Applicatives in Context

15.8.1 Why Applicatives?

There are multiple reasons for learning Applicatives, even if they do not provide any additional power over Monads. First off, as discussed in Lecture 13, the GHC standard library nowadays makes Applicative instances for all Monads compulsory. Thus a working Haskell programmer is bound to see lots of Applicative instances.

Secondly, you'll bump into Applicative operators a lot even in Monadic code. Expressions like `f <$> x <*> y <*> z` can be useful in many Monadic contexts. Also, since the `Traversable` type class is built in terms of `Applicative`, you'll often use applicative operations with it.

Thirdly, Applicatives are an excellent exercise in understanding functional design patterns. They marry the Functor pattern with the Monoid pattern, and Alternative brings in yet another Monoid-like dimension. Being able to efficiently work with Applicatives will make working with further abstractions like *monad transformers* or *lenses* easier.

Lastly, there are a couple of types that are Applicatives but not Monads. `Validation` is one example, and a very practical one at that. Without understanding of Applicative, we'd have no way to recognize and generalize operations over types like this. Another such type is `ZipList`.

15.8.2 Applicatives in the Wild

Even though we only looked at some very simple and concrete Applicatives in this lecture, there are plenty of Haskell libraries that use Applicatives for big things. Here are some examples.

The same sort of idea as our `Validation` Applicative has been implemented in the validation and either libraries.

There are multiple parser libraries that use Applicatives. For example, regex-applicative, optparse-applicative, yamlparse-applicative, json-stream, and so on.

15.8.3 Monads and Applicatives

So what's the relationship between an Monad and an Applicative? If an Applicative is also a Monad, the following laws hold:

```
pure             === return

fmap             === liftM

fmap f op        === do x <- op
                        return (f x)

liftA2           === liftM2
```

```
liftA2 f op1 op2 === do x <- op1
                        y <- op2
                        return (f x y)

op1 *> op2       === op1 >> op2

op1 <*> op2      === do f <- op1
                        x <- op2
                        return (f x)
```

When working in a `Monad`, you can freely mix `Applicative` and `Functor` with `Monad` operations. As an example, let's rewrite `mapM` until it only uses applicative operations, and thus get an implementation for `traverse`. This is our starting point:

```
myMapM op [] = return []
myMapM op (x:xs) = do y <- op x
                      ys <- myMapM op xs
                      return (y:ys)
```

GHCi tells us it only works for Monads:

```
Prelude> :t myMapM
myMapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

Let's apply the `pure === return` and the `liftA2` laws from above:

```
myMapM op [] = pure []
myMapM op (x:xs) = liftA2 (:) (op x) (myMapM op xs)
```

Ta-da! Now `myMapM` works for any `Applicative`:

```
Prelude> :t myMapM
myMapM :: Applicative f => (a -> f b) -> [a] -> f [b]
```

15.9 Quiz

What's the type of `x` in `liftA2 (&&) Nothing x`?

a. `Applicative f => f Bool`
b. `Applicative Bool`

c. `Maybe Bool`

How many elements does `liftA2 f xs ys` have when `xs` and `ys` are lists?

a. `length xs + length ys`
b. `length xs * length ys`
c. `min (length xs) (length ys)`

Which of these expressions is equivalent to `liftA2 f x y`? There might be multiple correct answers.

a. `f <$> x <*> y`
b. `f <*> x <*> y`
c. `f <*> x <$> y`
d. `fmap f x <*> y`
e. `pure f <*> x <*> y`

If `f :: a -> Maybe b` and `xs :: [a]`, which of these expressions has a type different from the others?

a. `fmap f xs`
b. `traverse f xs`
c. `map f xs`

For which `Applicative` do the expressions `pure x <* pure y` and `pure x <|> pure y` produce different results?

a. `Maybe`
b. `[]`
c. `Validation`

15.10 Exercises

- Set15

16 Lecture 16: Odds and Ends

This final lecture will go over some minor topics that didn't fit in anywhere else. You've finished all the hard parts of the course. Now it's time to sit back, relax, and enjoy some cool Haskell!

16.1 Testing with QuickCheck

One of the benefits of purity is that pure functions are easy to test: you don't need to set up any global state, you can just pass in arguments and check that the result is ok. In this section, we'll take a quick tour of the *property-based testing* library QuickCheck, which has also been used for checking that your answers to exercises are right on this course!

Let's look at testing a (faulty) implementation of `reverse`. You can find this and the following examples in the file `exercises/Examples/QuickCheck.hs`.

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

We can write an individual test case using the `===` operator from QuickCheck:

```
(===) :: (Eq a, Show a) => a -> a -> Property
```

```
propRevSmall :: Property
propRevSmall = rev [1,2] === [2,1]
```

We can ask QuickCheck to run them in GHCi:

```
*Examples.QuickCheck> quickCheck propRevSmall
+++ OK, passed 1 test.
```

So far so good. However this isn't really what QuickCheck was made for. QuickCheck was designed for *property-based testing* where you can state a property your code should have, and QuickCheck runs your code with randomized inputs, checking the property every time. What would be a simple property that a correct implementation of `reverse` has? Reversing a list twice should certainly give us back the same list. Let's write it out:

```
propRevTwice :: [Int] -> Property
propRevTwice xs = rev (rev xs) === xs
```

Our `Property` has an argument, which means that QuickCheck will generate random values and run the test. We can use the `verboseCheck` function to see which values are run. We can also give a parameter to the test ourselves if we wish to check a specific value.

```
*Examples.QuickCheck> quickCheck propRevTwice
+++ OK, passed 100 tests.
*Examples.QuickCheck> verboseCheck propRevTwice
Passed:
[]
[] == []

Passed:
[1]
[1] == [1]

Passed:
[-2,1,-1]
[-2,1,-1] == [-2,1,-1]
-- lots of output
+++ OK, passed 100 tests.
*Examples.QuickCheck> quickCheck (propRevTwice [1,2,3])
+++ OK, passed 1 test.
```

Even this property didn't catch the bug in our implementation. Let's try another one. Here's a property about how `rev (xs ++ ys)` behaves. You might want to take a moment to convince yourself that it should hold for a correct `rev` function.

```
propRevTwo :: [Int] -> [Int] -> Property
propRevTwo xs ys = rev (xs ++ ys) === rev ys ++ rev xs
```

Let's see if it holds for our implementation:

```
*Examples.QuickCheck> quickCheck propRevTwo
*** Failed! Falsified (after 5 tests and 3 shrinks):
[0,0]
[1]
[0,1,0] /= [1,0,0]
```

Finally, a failure! There's a bit to unpack here. First off, QuickCheck tells us the arguments with which the property failed: they are `[0,0]` and `[1]`. We can check this ourselves:

```
*Examples.QuickCheck> quickCheck (propRevTwo [0,0] [1])
*** Failed! Falsified (after 1 test):
[0,1,0] /= [1,0,0]
```

Next up, what does "after 5 tests and 3 shrinks" mean? One of the cool things about QuickCheck is that when it finds a failure, it tries some related values in order to find a nicer, smaller failure. We can see this in action with `verboseShrinking`, which prints out all the failures QuickCheck goes through:

```
*Examples.QuickCheck> quickCheck (verboseShrinking propRevTwo)
Failed:
[4,-1,-4]
[1,4]
[-1,-4,1,4,4] /= [4,1,-1,-4,4]

Failed:
[-1,-4]
[1,4]
[-4,1,4,-1] /= [4,1,-4,-1]

Failed:
[-4]
[1,4]
[1,4,-4] /= [4,1,-4]

Failed:
[4]
[1,4]
[1,4,4] /= [4,1,4]

Failed:
[0]
[1,4]
[1,4,0] /= [4,1,0]

Failed:
[0]
[0,4]
[0,4,0] /= [4,0,0]

Failed:
[0]
[0,2]
[0,2,0] /= [2,0,0]

Failed:
[0]
[0,1]
[0,1,0] /= [1,0,0]
```

QuickCheck went down from a counterexample of `[4,1,-1,4,4]` all the way to `[1,0,0]`. Pretty sweet!

16.1.1 Modifiers

Sometimes you need to limit the values QuickCheck generates. Your function might not work on all inputs, for instance? Let's try writing a test for `last`.

```
propLast :: [Int] -> Property
propLast xs = last xs === head (reverse xs)
```

```
*Examples.QuickCheck> quickCheck propLast
*** Failed! Exception: 'Prelude.last: empty list' (after 1 test):
[]
```

In this case, we can fix the test just by switching to another input type. QuickCheck defines the `NonEmptyList` type (not to be confused with `Data.List.NonEmpty`!), which is just a wrapper for a normal list. However, when generating values of `NonEmptyList`, QuickCheck won't generate empty lists.

```
newtype NonEmptyList a = NonEmpty [a]
```

```
propLastFixed :: NonEmptyList Int -> Property
propLastFixed (NonEmpty xs) = last xs === head (reverse xs)
```

```
*Examples.QuickCheck> quickCheck propLastFixed
+++ OK, passed 100 tests.
```

There are other modifiers like this, for example `Positive` for positive numbers, `NonNegative` for non-negative numbers, or `SortedList` for a sorted list. Here's an example of a more complex test. We check that the nth element of `cycle xs` is correct. Both of the modifiers are needed, since `!!` doesn't work with negative inputs, and `cycle []` is an error.

```
propCycle :: NonEmptyList Int -> NonNegative Int -> Property
propCycle (NonEmpty xs) (NonNegative n) =
  cycle xs !! n === xs !! (mod n (length xs))
```

16.1.2 Generators and `forAll`

Sometimes we need to limit the range of inputs to a test even further. As a simple example, here's a test that `Data.Char.toUpper` changes the character passed to it:

```
propToUpperChanges :: Char -> Property
propToUpperChanges c = toUpper c =/= c
```

```
quickCheck propToUpperChanges
*** Failed! Falsified (after 1 test and 1 shrink):
'A'
'A' == 'A'
```

Of course, it only changes *lowercase letters*. How can we write a test for that? There is no `Lowercase` modifier available that would work like `Positive` or `NonEmptyList`. We need to fall back to explicit generation of values using `forAll`:

```
propToUpperChangesLetter :: Property
propToUpperChangesLetter = forAll (elements ['a'..'z'])
          propToUpperChanges
```

```
*Examples.QuickCheck> verboseCheck propToUpperChangesLetter
Passed:
's'
'S' /= 's'

Passed:
'z'
'Z' /= 'z'
-- lots of output omitted
+++ OK, passed 100 tests.
```

Perfect! Let's look at the types to see what's going on here.

```
elements :: [a] -> Gen a
elements ['a'..'z'] :: Gen Char
forAll :: (Show a, Testable prop) => Gen a -> (a -> prop) ->
          Property
forAll (elements ['a'..'z']) :: Testable prop => (Char -> prop) ->
          Property
```

There are some new types here. A value of type `Gen a` is a generator for values of type `a`. We'll talk a bit more about `Gen` in the next section, but in this section you'll see a couple of functions that return `Gens` so that we can use them with `forAll`. The `elements` function is a generator that returns one of the elements of the given list at random, as you might have guessed.

The `Testable` type class is the same that the `quickCheck` function uses. It exists so that `quickCheck` can test types like `[Int] -> Bool -> Property` in addition to just plain `Property` values.

```
quickCheck :: Testable prop => prop -> IO ()
```

In addition, some simple types like `Bool` have a `Testable` instance, so that you can write tests using normal Haskell predicates instead of `===`:

```
listHasZero :: [Int] -> Bool
listHasZero xs = elem 0 xs
```

```
*Examples.QuickCheck> quickCheck (listHasZero [1,0,2])
+++ OK, passed 1 test.
*Examples.QuickCheck> quickCheck listHasZero
*** Failed! Falsified (after 1 test):
[]
```

Getting back to `forAll`, we can use `forAll` to write more complex tests. Here's a test that checks that `sort xs` has the same elements as `xs`. Note how we use `NonEmptyList` to guarantee that the `forAll` has some elements to pick from.

```
propSort :: NonEmptyList Int -> Property
propSort (NonEmpty xs) =
  forAll (elements xs) (\x -> elem x (sort xs))
```

16.1.3 Further with QuickCheck

We've only just scratched the surface of QuickCheck. Here are some pointers to some things you'll find useful when you start writing larger QuickCheck tests.

Sometimes the output from QuickCheck isn't quite verbose enough. You can add your own lines to the output by using the `counterexample` combinator:

```
counterexample :: Testable prop => String -> prop -> Property
```

As an example, let's add logging of the input to `rev` to `propRevTwo`:

```
propRevTwo' :: [Int] -> [Int] -> Property
propRevTwo' xs ys =
  let input = xs ++ ys
  in counterexample ("Input: " ++ show input) $
     rev input === rev ys ++ rev xs
```

```
*Examples.QuickCheck> quickCheck propRevTwo'
*** Failed! Falsified (after 4 tests and 5 shrinks):
[0]
[0,1]
Input: [0,0,1]
[0,1,0] /= [1,0,0]
```

As you might have guessed, `Gen` is a `Monad`. You can write your own generators by combining the generators defined by QuickCheck. You can check what your generators output using `sample`.

```
someLetters :: Gen String
someLetters = do
  c <- elements "xyzw"
  n <- choose (1,10)
  return (replicate n c)
```

```
*Examples.QuickCheck> sample someLetters
"yyyyyyyy"
"zzzzzzzzz"
"xxxxxxxxx"
"yyyyyyy"
"yyy"
"ww"
"xxxxxx"
"yyy"
"yyyyyyy"
"xxxxxxxxxx"
"y"
```

Closely related to generators is the `Arbitrary` type class. `Arbitrary` is how QuickCheck generates all those inputs automatically.

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
```

If you're writing tests for custom types, you either need to use `forAll`, or implement an `Arbitrary` instance. Here's what happens if you're missing an instance:

```
data Switch = On | Off
  deriving (Show, Eq)

toggle :: Switch -> Switch
toggle On = Off
toggle Off = On

propToggleTwice :: Switch -> Property
propToggleTwice s = s === toggle (toggle s)
```

```
*Examples.QuickCheck> quickCheck propToggleTwice
error:
    • No instance for (Arbitrary Switch)
        arising from a use of ‘quickCheck’
    • In the expression: quickCheck propToggleTwice
```

Here are the two ways to fix it:

```
*Examples.QuickCheck> quickCheck (forAll (elements [On,Off])
         propToggleTwice)
+++ OK, passed 100 tests.
```

```
instance Arbitrary Switch where
  arbitrary = elements [On,Off]
```

16.2 Phantom Types

Boo! There's a ghost in the type system! Let's have a look at what *phantom types* can do for you.

Phantom types are types that don't take any values. They are related to newtypes (see lecture 10) in that both are a way to add additional type checking without affecting the evaluation of the program at all.

Let's use phantom types to track which currency an amount of money is in. We define the phantom types `EUR` and `USD` (note how they don't have any constructors!), and the parameterized type `Money a` that doesn't use the type parameter `a` for anything. Then we can define two constants, one in euros and the other in dollars. You can find all of the code from this section in the file `exercises/Examples/Phantom.hs`.

```
data EUR
data USD
data Money currency = Money Double
  deriving Show

dollar :: Money USD
dollar = Money 1

twoEuros :: Money EUR
twoEuros = Money 2
```

Note how the type signatures for `dollar` and `twoEuros` are doing all the work here. An expression like `Money 1` has the polymorphic type `Money currency` if we don't restrict it to a more specific type. We give `dollar` and `twoEuros` more limited types explicitly. This is analogous to defining something like `one :: Int; one = 1`, since the constant `1` has the polymorphic type `Num p => p` but we give it a more restricted type.

```
*Examples.Phantom> :t Money
Money :: Double -> Money currency
*Examples.Phantom> :t Money 1
Money 1 :: Money currency
```

Now that we have some constants, we can write functions that operate on them. Let's start with a function `scaleMoney` that multiplies an amount of money by a number. The currency stays constant. Here too, the type signature is doing the work: without the type signature, Haskell would infer a type of `Double -> Money a -> Money b`.

```
scaleMoney :: Double -> Money currency -> Money currency
scaleMoney factor (Money a) = Money (factor * a)
```

```
*Examples.Phantom> :t scaleMoney 3 twoEuros
scaleMoney 3 twoEuros :: Money EUR
*Examples.Phantom> :t scaleMoney 3 dollar
scaleMoney 3 dollar :: Money USD
```

Next up: adding two amounts that are in the same currency. We get a nice type error if we try to add values in two different currencies.

```
addMoney :: Money currency -> Money currency -> Money currency
addMoney (Money a) (Money b) = Money (a+b)
```

```
*Examples.Phantom> :t addMoney dollar dollar
addMoney dollar dollar :: Money USD
*Examples.Phantom> :t addMoney twoEuros twoEuros
addMoney twoEuros twoEuros :: Money EUR
*Examples.Phantom> :t addMoney twoEuros dollar
error:
    • Couldn't match type ‘USD’ with ‘EUR’
      Expected type: Money EUR
        Actual type: Money USD
    • In the second argument of ‘addMoney’, namely ‘dollar’
      In the expression: addMoney twoEuros dollar
```

As before, the type signature is crucial. Here's the same implementation with an unrestricted type. Now we can add anything to anything!

```
addMoneyUnsafe :: Money x -> Money y -> Money z
addMoneyUnsafe (Money a) (Money b) = Money (a+b)
```

```
*Examples.Phantom> addMoneyUnsafe twoEuros dollar
Money 3.0
```

We can keep going with this approach, and define currency conversions. We define the type `Rate` that uses phantom types to track the currencies it's translating between. The types of `convert` and `invert` are restricted to have the properties we

want. There's also an unrestricted version of the convert function to let you compare the types.

```
data Rate from to = Rate Double
  deriving Show

eurToUsd :: Rate EUR USD
eurToUsd = Rate 1.22

convert :: Rate from to -> Money from -> Money to
convert (Rate r) (Money a) = Money (r*a)

invert :: Rate from to -> Rate to from
invert (Rate r) = Rate (1/r)

convertUnsafe :: Rate from to -> Money x -> Money y
convertUnsafe (Rate r) (Money a) = Money (r*a)
```

```
*Examples.Phantom> convert eurToUsd twoEuros
Money 2.44
*Examples.Phantom> convert eurToUsd dollar
error:
    • Couldn't match type ‘USD’ with ‘EUR’
      Expected type: Money EUR
        Actual type: Money USD
    • In the second argument of ‘convert’, namely ‘dollar’
      In the expression: convert eurToUsd dollar
      In an equation for ‘it’: it = convert eurToUsd dollar
*Examples.Phantom> convert (invert eurToUsd) dollar
Money 0.819672131147541
*Examples.Phantom> convertUnsafe eurToUsd dollar
Money 1.22
```

Note! The words `currency`, `from`, `to` and so forth in the previous examples are *just type variables*. There's nothing special going on with them. We could've as well given `invert` a type like `Rate a b -> Rate b a` without any change in type safety.

This approach that uses phantom types has clear benefits: giving us type errors for invalid code. Also, compared to defining lots of concrete types like `data MoneyEur = MoneyEur Double`, with phantom types we need to implement functions like `scaleMoney` and `addMoney` only once. Also, we're able to define polymorphic and reusable concepts like `Rate`. You can contrast this approach with the Boxing section of Lecture 7.

However, phantom types also have downsides. Without advanced tricks we can't really handle currencies that are defined at runtime (for example: reading an amount

from a user). It's also easy to end up in a place where you start needing language extensions like *Generalized Algebraic Datatypes*, *type families* and other *type-level programming* constructs. Eventually you're all the way in the world of *dependent typing*.

So what are good applications for phantom types? When you need to track some simple, but crucial information, that is known at compile-time. An example that's somewhat better than currencies is tracking whether inputs from the user have been sanitated to prevent attacks like SQL injection or cross-site scripting.

We can use the types `Input Safe` and `Input Unsafe` to track whether strings are safe for passing into the database or not. If our module only exports the `makeInput` function, and not the `Input` constructor, the type system ensures that any inputs must pass through the `escapeInput` function at some point before going into a database function like `addForumComment`.

```
data Safe
data Unsafe

data Input a = Input String

-- Public constructor function for Input, only allows constructing
-- Unsafe Inputs from Strings.
makeInput :: String -> Input Unsafe
makeInput xs = Input xs

-- Adds comment to the database.
addForumComment :: Input Safe -> IO Result
addForumComment = ...

-- We can combine inputs, but that won't change their safety
concatInputs :: Input a -> Input a -> Input a
concatInputs (Input xs) (Input ys) = Input (xs++ys)

-- Strip bad characters to turn an unsafe input safe
escapeInput :: Input Unsafe -> Input Safe
escapeInput (Input xs) = Input (filter (\c -> isAlpha c || isSpace
         c) xs)
```

16.3 Simultaneity

16.3.1 Parallelism

One of the great things about purity is that it makes *parallelism*, computing many things at the same time, very easy. Let's have a look at how we can do this in

Haskell. First off, we need to start a new GHCi that has parallel execution enabled. The simplest way is:

```
$ stack ghci --ghci-options "+RTS -N"
```

Next up, let's define a very naive version of the Fibonacci function (remember Lecture 1?) , enable performance statistics with `:set +s`, and see how long it takes to compute five values of the function:

```
Prelude> fib 0 = 1; fib 1 = 1; fib n = fib (n-1) + fib (n-2)
Prelude> :set +s
Prelude> map fib [29,29,29,29,29]
[832040,832040,832040,832040,832040]
(7.54 secs, 2,440,860,632 bytes)
```

Now let's bring in the module Control.Parallel.Strategies that defines ways to evaluate values in parallel. We'll use the `parList rseq` strategy to evaluate all the elements of the list to WHNF, *in parallel*.

```
Prelude> import Control.Parallel.Strategies
Prelude Control.Parallel.Strategies> withStrategy (parList rseq)
         (map fib [29,29,29,29,29])
[832040,832040,832040,832040,832040]
(4.80 secs, 488,531,384 bytes)
```

That's almost 2 times as fast, on the 2-core machine that this example is being run on. Pretty nice. The coolest thing here is that we were able to define the *computation* (`map fib ...`) completely separately from the *evaluation strategy* (`parList rseq`), separating the *what to compute* from the *how to compute*.

16.3.2 Concurrency

Computer science makes the distinction between parallel and *concurrent* computations. Parallel computations are those that just run separate independent computations in parallel (in other words, parallelism is *pure*). Concurrent computations are those where there are multiple interacting threads of computation. Concurrency usually involves threads, locks, messages and deadlocks.

In addition to great tooling for parallelism, Haskell also has good tooling for concurrency via *threads*. Since concurrency is all about side-effects, concurrent computations happen in the `IO` Monad. The classic example of threading is two

threads, one printing a stream of As and the other a stream of Bs. Here it is in Haskell:

```
printA :: IO ()
printA = putStrLn (replicate 40 'A')

printB :: IO ()
printB = putStrLn (replicate 40 'B')

concurrency :: IO ()
concurrency = do
  forkIO printA
  forkIO printB
  return ()
```

```
Prelude Control.Concurrent> concurrency
AABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
```

The operation `forkIO :: IO () -> IO ThreadId` takes an IO operation and starts running it in the backgroud. It produces a `ThreadId` that can be used to e.g. terminate the thread.

If we want to add actual communication between threads, we can use abstractions like `MVar` (a mutable thread-safe variable) or `Chan` (a queue).

Here's a simple example where one thread writes a value to an `MVar` and another one waits for them and prints them. An `MVar` works like a mailbox: it is either empty or full. Calling `takeMVar` on an empty box waits for the box to get filled (with a `putMVar`). Symmetrically, trying to `putMVar` into a full box waits until the box is empty.

```
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
newEmptyMVar :: IO (MVar a)
```

```
send :: [String] -> MVar String -> IO ()
send values var = mapM_ (putMVar var) values

receive :: MVar String -> IO ()
receive var = do val <- takeMVar var
                 print val
                 -- loop unless at last value
```

```
                when (val/="end") (receive var)

concurrency2 :: IO ()
concurrency2 = do
  var <- newEmptyMVar
  forkIO (send ["hello","world","and","goodbye","end"] var)
  forkIO (receive var)
  return ()



Prelude Control.Concurrent Control.Monad> concurrency2
"hello"
"world"
"and"
"goodbye"
"end"
```

16.4 Exercises

- Set16a: QuickCheck
- Set16b: Phantom types
- No exercises for parallel or concurrent Haskell, sorry!

16.5 Where to Go From Here?

Congratulations! You’ ve reached the end of this two part course on Functional Programming in Haskell. What next? You definitely know enough Haskell to keep learning on your own. The online Haskell community is very friendly and there are lots of blog posts and other content explaining advanced techniques and features. You can find lots of interesting stuff by following for example:

- /r/haskell on reddit
- `#haskell` on libera.chat
- Haskell Weekly
- StackOverflow

Just keep writing Haskell, studying things (like libraries and tools) when you bump into them, and slowly accumulate experience. Lots of the things you learn with Haskell will be transferable to other languages like TypeScript, Elm, Rust or F#.

Finally, here is an incomplete list of things that got left out of this course, but are worth looking into:

- Language features

    - Modules
    - Lazy patterns (~ patterns) and @ patterns. See e.g. A Gentle Introduction to Haskell.
    - Language extensions like `MultiParamTypeClasses`, `ViewPatterns` etc. This is one good guide
    - The `fix` function
    - The Foreign Function Interface for calling C code from Haskell
- Abstractions
    - Monad transformers: RWH, Wikibook
    - Free monads (advanced topic): blog
    - Lenses (advanced topic): tutorial glassery
    - Bartosz Milewski covers lots of intermediate and advanced topics on his blog
- Tooling
    - Using Cabal and Stack to build your own projects
    - Profiling: RWH, GHC
    - Hlint
- Libraries
    - Parsec (parsing): RWH
    - Scotty (simple web framework), Aeson (json): blog
    - Servant (fancy web framework with phantom types)
- Category theory
    - Many Haskell abstractions are based on Category theory
    - Category theory can be a valuable source of new ideas for programming
    - Category theory can feel intimidating, so it's good to know that you can get on fine without it
    - Bartosz Milewski has lots of good material, for example Category Theory for Programmers
    - The Haskell Wiki and Wikibook have sections on category theory

16.6 Acknowledgements

This course was made possible by Nitor who donated hours and hours of Joel's working time for this project. Thank you!

Thanks to the whole Haskell Mooc team, especially

- John Lång for help on the material
- Antti Laaksonen for setting up the course and helping with arrangements
- Topi Talvitie for the exercise checking infrastructure

Thanks to all the students who patiently waited for part 2 and reported errors in the material & exercises!