Haskell Handbook

Shahzeb Asif

June 17, 2017

References

LYAH

Miran Lipovaca's great book:

LYAH

UChicago CMSC-16100

Stuart Kurtz's lecture notes:

CMSC-16100

adit.io

Great explanation of functors, applicatives, and monads:

adit.io

Basics

Basic Properties

Haskell is a **lazy**, **statically typed** language. It can infer types as well.

Comments in Haskell are indicated by preceding `--` for single-line comments and `\{- text -\}` for blocks.

Basic Operators

- Math: `+ * - /`
- Logic: `&& || not`
- Ordering: `== /= > <`

Basic Functions

- **succ** returns the next item in a sequence.
- **pred** returns the previous item.
- **min** returns the min in the input parameters.
- **max** returns the max in the input parameters.

GHC

To compile the file we use `ghc --make file.hs`. There are some useful flags for ghc:

- `-O2` optimises the executable.
- `-dynamic` dynamically linkes the libraries and helps keep the executable size down.

GHCi

```
> :t (True, 2)
(True, 'a') :: (Bool, Char)

> :t length
[a] -> Int
```

:t in GHCi

:t returns the type signature:

- The notation above states that `length` takes a list and returns an `Int`
- Also note that the a in the output is called a type variable and it indicates that length is a **polymorphic function** which allows it take different types of input.

```
> :info Bool
data Bool = False | True            -- Defined in 'GHC.Types'
...
```

`:info` in GHCi

:info returns info about the type or type class.

```
:k Maybe
Maybe :: * -> *
```

`:k` in GHCi

:k in ghci is called kinds and can be used to find information about types.

- The example states that `Maybe` takes in a concrete type and returns a new concrete type.

Functions

Arrow

The **arrow** `->` operator is from lambda calculus. The scope of the arrow `->` extends all the way to the right.

```
\x -> \y -> sqrt(x^2 + y^2)
\x y -> sqrt $ x^2 + y^2
```

Pattern

Pattern matching is used to conform input to a defined pattern. An example of this might be using the `:` operator to partition an input list into two parts.

In the body, it has a if/elif sort of structure. If the first statement occurs then the one later will not. Because of this property, it is better to specify the specific catches first and the general ones later. If you define a function with limited catches and no general case then it could fail.

```
lucky :: (Integral a) => a -> String
lucky 7 = "Lucky!"
lucky x = "Nah"

myLength :: (Num a) => a -> a
mylength [] = 0
mylength (first:rest) = 1 + myLength rest
```

Pattern Matching Examples

Note that you cannot use `++` in pattern matching as it is not a data constructor. That's the key to understanding pattern matching. We use it to resolve data types into their underlying data constructor and generic field.

`@` is something we can use to do pattern match but still keep reference to the original item. A simple example of this is `x@(y:ys)`.

Note that to do pattern matching you should put your pattern in brackets. This is tied into how pattern matching works by deconstructing data types.

Case

Pattern matching is actually just a different syntax for `case`.

```
myLen2 :: (Num a) => [b] -> a
myLen2 xs = case xs of [] -> 0
                       (_,r) -> 1 + myLen2 r
```

Guards

Guards are more like traditional if/elif statements. We follow the first logic check is `True`. We also have a catch-all statement at the end to catch anything missed by our logic checks.

```
abs x
    | x > 0 = x
    | x == 0 = 0
    | otherwise = -x

abs x = if x > 0 then x
        else if x == 0 then 0
        else -x
```

Equivalent Functions

The two implementations of the absolute value function above are essentially equivalent. The `|`, guards, syntax is just syntactic sugar for the traditional if/else statements.

```
amIFat :: (RealFloat a) => a -> a -> String
amIFat weight height
    | bmi <= 18.5 = "Underweight"
    | bmi <= 25.0 = "Normal"
    | bmi <= 30.0 = "Fat"
    | otherwise = "Whale"
        where bmi = weight/height ^ 2

recArea :: (Num a) => [(a, a)] -> [a]
recArea listRec = [area x y | (x, y) <- listRec]
    where area x y = x * y
```

Examples of Guards

There are two other important details with guards: `otherwise` and `where`.

The catch-all statement we mentioned before is the **otherwise** keyword in Haskell. It is usually located after all the checks in a guard block.

Another keyword called **where** is often used with guards to calculate a value once and reduce redundancy. `where` allows us to create local variables and functions.

Let

let and **in** are are to `where` as `case` is to pattern matching.

```
recArea2 :: (Num a) => [(a, a)] -> [a]
recArea2 listRec =
    let area x y = x * y
    in [area x y | (x, y) <- listRec]
```

Using `let/in`

The advantage of `let` is that you can use it in more situations:

```
let perimeter x y = 2*x + 2*y in (perimeter 10 15, perimeter 1 2)

recArea3 :: (Num a) => [(a, a)] -> [a]
recArea3 listRec = let area x y = x * y in [area x y | (x,y) <- listRec]

recArea4 :: (Num a) => [(a, a)] -> [a]
recArea4 listRec = [area x y | (x,y) <- listRec, let area x y = x * y]
```

Examples of `let/in`

Recursion

```
myMaximum :: (Ord a) => [a] -> a
myMaximum [] = 0
myMaximum (n:[]) = n
myMaximum (h:t) = max h myMaximum t
```

Recursive `max` Function

Recursion is incredibly important in Haskell. The most basic form of recursion is usually in splitting a list into its first item and the rest of the list. You can solve a lot of problems using that basic pattern.

The most important thing to remember is that recursion is best learned with practice.

Higher Order Functions

Curried Functions

Functions like the `max` function aren't what they seem. `max` takes a single parameter and returns a function that compares another parameter with the original parameter.

A simple example (LearnYouAHaskell) is `> max 2 3`. This function will first return a function that takes a number as an argument and compares to three then it will call that new function with 3.

That seems confusing so let's walk through it.

1. First `max` takes in `2` as an argument and returns a new function. This new function takes in another integer as an argument and compares it only to `2`.
2. The new function is called with `3` which is compared to `2`.

This concept is crucial to understanding a key property of functional languages. Their ability to have partial function is called `currying`. As it turns out, all functions in Haskell only have one argument. Functions that need more than one argument are called again and again in the way we discussed above.

Another consequence of curring and Haskell's laziness is that you can call functions with too few variables and no one will complain because you'll just get a function back as long as you don't ask Haskell to display the result. You could then call this returned partial function with another argument to get your final answer.

Although it's important to understand currying, it's still easier and more practical to think of functions in their more traditional programming form.

Functions can also be passed as arguments but they still have to be defined explicitly in the function type.

```
applyThrice :: (a -> a) -> a -> a
applyThrice f x = f (f (f x));

myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
myZipWith _ [] _ = []
myZipWith _ _ [] = []
myZipWith f (h1:t1) (h2:t2) = (f h1 h2):(myZipWith f t1 t2)
```

Functions as Arguments

Sections

Sections are binary functions that only require and argument but it doesn't matter it if was supposed to be a left hand or right hand argument. The reason for this is currying.

This property is useful to make more general functions.

```
> (+3) 7
10

> (3+) 7
10;


square = (^2)
```

Useful Functions

- **map** takes a function and applies it to all items in a list:

```
> map (head) ["test", "function"]
tf

> map (3-) [1, 2, 3]
[2, 1, 0]

> map (-3) [1, 2, 3]
ERROR
```

Mapping Over A List

- **filter** is useful for getting rid of values from a list:

```
larDivisible :: (Integral a) => [a] -> a -> a
larDivisible l d = last (filter p l)
    where p x = (mod x d) == 0
```

Filtering The Largest Divisor

- **takeWhile** takes elements from a list while a condition is satisfied

These three functions are incredibly powerful. Learn them extremely well.

Infix

Infix notation is very useful for certain functions that area easier to understand as infix functions. Functions can be called in infix notation by using backticks.

```
> div 4 2
2

> 4 `div` 2
2

> (div 4) 2
2

> (`div` 2) 4
2

> (flip div 2) 4
2
```

Infix Notation

Note how we used infix notation and the **flip** function to change which argument was used first. Generally, infix notation is cleaner than the `flip` function.

Lambdas

Lambdas are used to create anonymous functions. Anonymous functions are rstricted functions that are used for very simple tasks. Their advantage is that they have minimal overhead and can be embedded in more complicated statements.

In Haskell, they are indicated by a
and wrapped in brackets:

```
;zipWith (\x y -> x*2 + 5^(2*y)) [1, 2] [4, 6];
```

Lambdas can also pattern match but for only one case.

Folding

Folding is another common technique in Haskell. It allows you to iterate through an object in a fashion. You call a fold with a binary function, a starting value for an accumulator and a list and what it does is run through each item in the list applying the binary operator to the item from the list and the accumulator.

- **foldl** starts folding from the left:

```
> foldl (+) 0 [1, 2, 3, 4, 5]
15

accumulator (operation) item
```

`foldl` Example

- **foldr** folds from the right: `item (operation) accumulator`
- **foldl1** and **foldr1** work like regular folds but they just take the left-most and right-most values, respectively, to use as the accumulator.
- **scanl** and **scanr** are like foldl and foldr but they return a list after every pass instead of just a single final value.
- **scanl1** and **scanr1** are analogous to foldl1 and foldr1.

```
> length (takeWhile (< 1000) (scanl1 (\x y -> x + sqrt y) [1..]))
```

How many natural numbers does it take for their square root to exceed 1000?

$

Use `$` to calculate all to the right of the `$`:

```
> (10 /) 5 + 2
4

> (10 /) $ 5 + 2
1.428...
```

Example of `$`

Function Composition

The `.` operator is used for function composition. This is like the math property: $(f \circ g)(x) = f(g(x))$ This is used to simplify code.

Point-Free Style

This is a way to reduce redundency in functions:

```
> negative 5
-5

negative x = negate (abs x)

vs.

negative = negate . abs
```

Point-Free Statements

We can eliminate redundant numbers/variables from both sides.The mathematical word for this is **eta-reduction**.

There is one important thing to remember. If you pattern match an argument, you must use it in the right-side of the statement. Note this in the last two examples in the code above.

Types

As previously mentioned, to get the type of something in GHCi, use `:t`.

```
> :t 'a'
'a' :: Char
```

The `::` means 'type of'.

Atomic and Real Types

- **Int** are signed 32 bit numbers.
- **Integer** are not bounded.
  - Use `Integer` sparingly because they are quite slow.
- **Float** have single precision.
- **Double** have double precision.
- **Bool** can be True or False.
- **Char**

Lists

Operations

- `:` attaches an item to the front of a list:
  - This is instantaneous.
  - This operation is sepcial as well. It's actually a "synonym" for the `cons` function from Lisp. It is used in the construction of a list and is a data constructor. That's why it is used a lot in pattern matching.
- `++` concantanates two lists:
  - **Caution:** this operator runs through the whole left list.
- `!!` is used to index a list:

```
> "hello" ++ " world"
"helloworld"

> 1:2:[3,4]
[1,2,3,4]

> "Test" !! 1
'e'
```

List Operations

Useful Functions

```
> elem 2 [1,2,3,4]
True

> take 2 [1,2,4,5,7]
[1,2]

> [1,2..10]
[1,2,3,4,5,6,7,8,9,10]

> [1,3..10]
[1,3,5,7,9]

> replicate 3 "test"
["test", "test", "test"]
```

Functions on a List

- **head** returns the first item in a list.
- **tail** returns all items excluding the first.
- **last** returns the last item in a list.
- **init** returns all items excluding the last.
- **length**
- **null** checks if a list is empty.
- **reverse**
- **take** extracts n items from the start of a list.
- **drop** deletes n items from the start of a list similar to **take**.
- **maximum**
- **minimum**
- **sum**
- **product**
- **elem** tells you if an item is in a list.
- **..**
  - **cycle** pretends a list is an infinite circle and the tail is connected back to the head.
- **repeat** creates an infinite list out of a parameter.
- **replicate** returns n copies of a list inside a list.

List Comprehensions

List comprehensions work differently in Haskell than Python. E.g. `> [x | x <- [1,2,3,4,5], x < 5]`. The list above can be read as: create a new list where x is an element from a list such that $x < 5$.

We can have multiple conditions at the end as well: `> [x | x <- [1..10], x /= 2, x < 5]`

```
myLength s = sum [1 | _ <- s]
removeNoneUppercase s = [c | c <- s, elem c ['A'..'Z']]
rightTriangles  = [(a,b,c) | c <- [1..10], b <- [1..c], c <- [1..b], a^2 + b^2 == c^2]
```

Examples form LearnYouAHaskell

Note that in the `rightTriangles` above, the reason the values for b exclude all values above c is to avoid repeats.

You can even have list comprehensions inside list comprehensions.

Tuples

Basic Properties

Tuples are immutable. Tuples, unlike lists, do not have to be homogenous. A tuple decides its type based on the type of the items inside and their index as well. The following pairs, `(1, 2) (1, 'a')`, have different types:

Useful Functions

- **fst** retuns the first item in a tuple pair.
- **snd** return the second item.
- **zip** combines two lists into pairs limited by the smaller list.

```
> zip [1, 3, 45] ['a', 'c', 'f', 'b']
[(1,'a'), (3,'c'), (45,'f')]
```

Using `zip`

Typeclasses

They're almost like properties that your values or functions are a part of. Typclasses define a functions behaviour.

```
> :t (==)
(==) :: (Eq a) => a -> a-> Bool
```

TypeClasses

Considering the example above: The output tells us that the two parameters `a` must be of the same type. This is kind of like forcing a comparison between apples and apples and not allowing any apples to orange comparisons.

This is specified by the typeclass prior to the `=>`.

Basic Typeclasses

- **Eq** support `==` or `/=`.
- **Ord** support `<`, `>`, `>=`, `<=`.
- **Show** can be turned into strings.
- **Read** turns strings to a type that supports `read`.
  - Note that read alone won't work. `read` decides the return type by inferring the return type from other operators.
  - `> read "4"` gives `ERROR`.
  - `> read "4" :: Int` gives `4`.
- **Reads** returns an empty list if it fails to read.
- **Enum** are things that use succ and pred.
- **Bounded** are things that have an upper and lower bound.
- **Num** are things that can act like numbers.
- **Integral** only include whole numbers and is kind of a subclass of Num.
- **Floating** includes floats and doubles.

Custom Types

Data Types

We can define our own data types using the **data** keyword The way we define data type is by giving it a name and all the other data types it could be.

```
data Bool = False | True

data Shape = Circle Float Float Float | Rectangle Float Float Float Float
             deriving(Show)
```

Simple Data Types

IN the example above, `Circle` and `Rectangle` are not types. They are **data constructors**.

Data constructors return a value of a given type. The contructors can take types themselves and we call those fields. The cool thing is that when we used things like `4` or `False` we were using data contructors with no fields.

The types and constructors must be capitals.

deriving is a keyword used to add your types to typeclasses. There is a little bit of magic going on here. Note that `deriving` does not work with custom typeclasses. `deriving` creates an `instance` of your data type of the specified typeclass. This allows your data type to have the behaviour specified by the typeclass.

```
sarea :: Shape -> Float
sarea Circle _ _ r = pi * (r ^ 2)
sarea Rectangle x1 y1 x2 y2 = (x2 - x1) * (y2 - y1)

data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)

sarea (Rectangle (Point x1 y1) (Point x2 y2)) = abs $ (x2 - x1) * (y2 - y1)

baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRect :: Float -> Float -> Shape
baseRect w h = Rectangle (Point 0 0) (Point w h)
```

Examples from LearnYouAHaskell

Record Syntax

There's another cool way to make data types more organized. We can assign each field a name. `data Person = Person {firstName :: String, lastName :: String, age :: Int}` This creates functions called `firstName`, `lastName`, and `age`. These helper functions take in a `Person` and return their fields.

```
> let me = Person "Shahzeb" "Asif" 20
Person {blah blah...}

> firstName me
"Shahzeb"
```

Records

Using record syntax also allows us to be more flexible and explicit in creating people:

```
Person {firstName = "Bob", lastName = "Tim", age = 100}
```

Type Constructors

Type constructors are kind of like types but they allow you to change the type of the value returned based on the value passed in to a data constructor.

```
data Maybe a = Nothing | Just a
```

`Maybe`

`Maybe` is an example of a type constructor. The `Just` keyword we used is nothing more than a data constructor. It will return a new value of type `Maybe Int` or a `Maybe Char`. The

Type constructors are like meta types. They're useful when their application could be applied to any other type.

Do not add typeclass constraints in data declarations because they clutter up other functions but imposing unnecessary restrictions.

```
data Vector a = Vector a a a deriving (Show)

vplus :: (Num t) => Vector t -> Vector t -> Vector t
vplus (Vector i j k) (Vector l m n) = Vector (i+l) (j+m) (k+n)
```

Example from LearnYouAHaskell

Note how in the example above, the function is restricting the types to `Vector Num` but the data constructor doesn't do any restricting. Also note that the type signature of `vplus` uses `Vector t` and not `Vector t t t`. This is because we use types in the signature and not the values.

To reiterate, the left-side of the = is the type constructor or the type of the returned value. The right-side of the = is the data constructor which returns a value of a certain type

Concrete types are any types that can have actual values, e.g:

- `Int`
- `Maybe Int`
- `Char`
- etc.

This StackOverflow answer answer helps clarify details here:

Instances

We've previously talked about `instances` and how they're created with `deriving`. Here are some more examples of `instances` and what they allow us to do.

```
-- Char is an instance of Ord because we can compare chars.
> 'a' > 'b'
False

-- Now we can check if two Points are equal.
data Point = Point Float Float deriving (Show, Eq)

(Point 0 0) == (Point 1 0)
False

(Point 0 0) == (Point 0 0)
True

elem (Point 1 0) [(Point 0 0), (Point 1 1), (Point 2 2)]
False
```

Examples

Newtype

newtype is reserved for a special case where a data type only has one possible form.

It's like a virtual type that's fast to work with for this special case. It should be treated as a normal type for all practical purposes. The restriction is that newtype can only have one constructor and one field.

One key difference between newtype and data is that with newtype, Haskell doesn't have to pattern match the data constructor or the value field because there can only be one.

```
data Cool = Cool {getCool :: Bool}

helloMe :: Cool -> String
helloMe (Cool _) = "hello"

> helloMe $ Cool True
"hello"
> helloMe undefined
"*** Exception:......

newtype Cool = Cool {getCool :: Bool}

> helloMe $ Cool True
"hello"
> helloMe undefined
"hello"
```

Differences between `data` and `newtype` from LearnYouAHaskell

In the second example using `newtype`, Haskell doesn't bother to check if undefined and Cool match, it just does what it needs to do.

Type Synonyms

We use the **type** keyword to describe synonyms for existing types: `type` does not create anything new and it cannot be applied to more than one item, i.e. it is **not** an alternative to data constructors.

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name, PhoneNumber)]

inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook n pn pb = elem (n, pn) pb
```

Another great example from LearnYouAHaskell:

The type signature for `inPhoneBook` tells a lot about the function. If we hadn't used `type`, the type signature could instead be `inPhoneBook :: String -> String -> [(String, String)] -> Bool`. This type signature gives a lot less information about the function.

Common Types

Maybe

Maybe is used for error checking:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord)
```

Either

Either is a type constructor that allows you to have two different types:

```
data Either a b = Left a | Right b
                  deriving (Eq, Ord, Read, Show)
```

This type is used for error checking that's more detailed than `Maybe`. If there is an error then we can return using `Left` and `Maybe a` string but if it was successful then we can return using `Right` and the return value.

This way allows you to fail at more than one point which is useful for debugging.

Association Lists

Association lists are just key-value pairs:

```
newtype Assoc a b = Assoc [(a,b)]
```

Recursive Structures

- This is a slightly weird way to create types that split easily:

```
data List a = Empty | Cons a List a

> Cons 3 (Cons 2 (Cons 4 Empty))
Cons 3 (Cons 2 (Cons 4 Empty))
```

Our Own List

The type above is our very own list. The way it works is by stringing along some time a until Empty is reached at the end. The `Cons` operator is equivalent to `:` in normal lists. We pattern match using the `cons/:` operator.

Fixity Declarations

We can define our own infix operator using the keywords **infix, infixr, infixl**. We use infixr because our operator is right-associativity.

```
infixr 5 :-:
data List a = Empty | a :-: (List a)
    deriving (Show, Read, Eq, Ord)
```

Our Own :

The 5 means that it takes a lower precedence than `+` or `*`.

Type Examples

We will implement a binary search tree. This is a simple tree that places values less than current node to the left and values greater to the right. This eventually results in a tree where all the values to the left are lower and all the values to the right are higher.

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)
    deriving (Show, Read, Eq)

singleton :: a -> Tree a
singleton a = Node a EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a lt rt)
    | x == a = Node x lt rt
    | x < a = Node a (treeInsert x lt) rt
    | x > a = Node a lt (treeInsert x rt)

treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a lt rt)
    | x == a = True
    | x < a = treeElem x lt
    | x > a = treeElem x rt
```

Tree from LearnYouAHaskell

Study this module and note an efficient way to insert items from a list: `foldr Tree.treeInsert EmptyTree [1, 2, 3, 4, -1`

Typeclasses

We've already discussed typeclasses but to reiterate: typeclasses affect the way types behave. They are **NOT** like classes from other languages.

The **class** keyword is used to define a new typeclass.

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y :: not (x /= y)
    x /= y :: not (x == y)
```

Eq

Eq is a cool typeclass. Note how the functions are recursively defined. This is because they are just the default definitions. To use this class, an instance just has to define one of the two functions. That way, the other one automatically works.

Instance

We can create our own instances of a typeclass which is necessary for custom typeclasses. This is best shown with an example.

```
data trafficLight = Red | Yellow | Green

instance show trafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

Custom TypeClass

Note the **instance** keyword to create our own instance of a typeclass. Also note that the function show was effectively overwritten for this type.

We would also define instances of different types for a typeclass.

```
class YesNo a where
    yessno :: a -> Bool

instance YesNo Integer where
    yesno 0 = False
    yesno _ = True
```

Functor is a typeclass for mapping over things. It will be explained in much more detail later.

```
class MyFunctor f where
    myfmap :: (a -> b) -> f a -> f b

instance myFunctor [] where
    myfmap f [] = []
    myfmap f (x:xs) = (f x):(myfmap f xs)

instance MyFunctor Tree where
    myfmap f EmptyTree = EmptyTree
    myfmap f (Node v lt rt) = Node (f v) (myfmap f lt) (myfmap f rt)
```

MyFunctor

This is a simple implementation of "Functor". Note how different implementations are defined for `myfmap` in each instance.

Functors

Functor is a typeclass with only one method: `fmap`. `fmap` takes in a function and uses it on a value in a given **context**. By context, I mean that there is some overhead involved in dealing with and storing the value, e.g. `Maybe`, `Either`,...

The function passed in must only take one parameter. To put it simply: **a functor applies a simple function to a value in a context and then returns a new value in the same context**.

```
> fmap (+3) (Just 3)
Just 6

> fmap (+3) [1, 2, 3]
[4, 5, 6]
```

Using `fmap`

Also note that when people refer to functors, they mean the specific instances of the typeclass Functor, i.e. things you can map over.

Laws

There are two laws that all instances of Functor must obey.

1. The first law is that mapping id over a functor must return the functor itself.
2. The second law is that mapping using a composition of two functions must be the same as mapping using the two functions separately.

```
-- First Law
> fmap id [1,2]
[1,2]

-- Second Law
;> fmap (2*) $ fmap (3+) [1,2]
[8,10]
> fmap ((2*) . (3+)) [1,2]
[8,10]
```

Laws of Functors

Applicative

This is another class that fills a gap in using functors stored in `Control.Applicative`.

If we apply a partial function to a functor which will need another argument later, we can't simply use fmap again.

`> fmap [5,6,7] (fmap (*) [1,2,3])` gives `ERROR`

We can track the reason this doesn't work.

1. After first mapping `(*)` we end up with `[1*, 2*, 3*]`
2. But now we try to map functions in a context using fmap but fmap only works with functions that work outside of contexts. This causes an error.

For these cases, we have the Applicative typeclass:

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

`Applicative`

- `pure` takes a concrete type and returns it in a new context.
- `<*>` is a function that we can pass a type with a function already applied to it, i.e. like `[1*, 2*, 3*]` from above and it will apply it to the argument:
- `<$>` is another syntax replacement for fmap. It is identical to fmap.

To put it simply: **applicatives take a function in a context and apply it to a value in a given context**.

```
> let t = fmap (*) [2,3]
> t <*> [3]
[6,9]

> (*) <$> [1, 2, 3] <*> [4]
[4,8,12]

> (*) <$> [1, 2, 3] <*> 4
ERROR

> (*) <$> [1, 2, 3] <*> pure 4
[4, 8, 12]

> (*) <$> Just 3 <*> Just 6
Just 18

> (++) <$> getLine <*> getLine
```

Examples of `Applicatives`

`IO`, `[]`, and `Maybe` are some of the instances of Applicative functors.

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something

instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]

instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```

Some `instances` of `Applicative`

Monads

Definition

Monad is just a typeclass that has two key functions: `return` and `(>>=)`.

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

`Monad`

- The `return` function will take a type and return the monad compatible version of it, e.g. It might take in an `Int` and return a `Maybe Int`. It is very similar to `pure` in applicatives.
- The `>>=` operator is pronounced bind. It is used to bind operations together.

To get a better sense of `Monads`, let's look at `Maybe`.

```
instance Monad Maybe where
    return x = Just x
    Just x >>= f = f x
    Nothing >>= _ = nothing
```

`Maybe`

`return` is easy to understand and even easier after understanding bind.

What bind ends up doing is it feeds the result of the previous operation to the next. This is used to create chains of commands.

To put it simply: **Monads allow you to perform complex computations in contexts**.

Looking at the definition of bind in `Maybe`, it will open up a `Maybe x` and then apply a function but this function MUST return a `Maybe x`.

The easiest way to understand how bind works is to work through an example. This example is derived from the CMSC-16100 lecture notes.

```
father :: Person -> Maybe Person
gfather p = father p >>= father
```

First, note that `father` is some function that may return a persons father. It takes in a type `Person` and returns a `Maybe Person`

Using `father` and the Monad instance of Maybe, we can easily define gfather to find a person's grandfather.

The father of a `Person p` is found. This will return a `Maybe Person` type which will then be fed into the next call for father.

This results in father being called successively. And if `father p` returned a `Nothing` then the whole thing would return `Nothing`.

Every monad is also a `Functor`.

Laws

There are three laws that every monad instance should follow:

1. **left identity**: `return a >>= f  ==  f a`
2. **right identity**: `ma >>= return  ==  ma`
3. **associativity**: `(ma >>= f) >>= g  ==  ma >>= (\x -> f x >>= g)`

Functions

There are other functions used with Monads apart from `return` and `bind`.

- `>>` is used when we just want to toss the results.
- `=<<` is just the bind operator reversed.
- `>==>` and `<==<` are a little more complicated so look them up on typeclassopedia.

Do

The `do` blocks used in IO are actually just syntactic constructs for the binding operators. The do block allows us to do imperative style programming but using functions.

Note that imperative style does not mean imperative.

```
return 1 >>= (\x ->
    return (x*x) >>= \(x ->
        return (x+1) >>= (\x ->
            return x)))
```

```
do
    x <- return $ 1
    x <- return $ x * x
    x <- return $ x + 1
    return x
```

Equivalent Blocks from CMSC-16100 Notes

Let's go over exactly what happens.

We know that `>>=` takes in two arguments: a Monad type and a function that returns a Monad type.

So in the first line with `return 1 >>= (\x ->`, `return 1` gives us a `Monad Int` which then gets fed into another anonymous function. And it works out because this new function takes in a regular type and returns another Monad type.

It turns out that the `<-` notation is just the computation flow. The variable on the left of the `<-` is just the argument for the next anonymous function that takes a regular type and returns a Monad type.

This is also why the last line in a do block cannot have the `<-`. Because bind must return a Monad type.

Common Monads

Maybe

Maybe is used an awful lot in Haskell. It's used largely inplace of exceptions. It's better to return a `Maybe a` than raise exceptions constantly.

The real strength of `Maybe` is in chaining `Maybe` statements which lets you do many things and has built in error checking. If one thing were to fail, the whole block would fail.

Get

`Get` is useful for byte operations. Here's an example adapted from the Haskell wiki:

```
import qualified Data.ByteString.Lazy as BL
import qualified Data.Binary.Get as BG

main = do
    content <- BL.getContent
    print $ BG.runGet myParser content

myParser :: BG.Get String
...
```

Byte Parser

`runGet` is a function that takes a `Get a` and runs it on a bytestring. Our parser does actually operate on the content just in a slightly non-obvious way.

Modules

Prelude is the default module but you can import others. Below are all the ways you can import modules:

- `import Data.List`
- `import Data.List Data.Map Data.Set`
- `import Data.List (nub, sort)`
- `import Data.List hiding (nub)`

Importing modules this way puts them in the global namespace but if you want to specify which module they're a part of then we can use the **qualified** keyword:

- `import qualified Data.List`
- `import qualified Data.List as DL`

Basic Modules

Data.List

```
> intercalate "test" ["This", "is", "person"]
"Thistestistestperson"

> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]

> all (>3) [2..4]
False

> any (>3) [2..4]
True

> span (`elem` ['A'..'Z']) "ROOminG"
["ROO", "minG"]

> break (== 'a') "Blooyay"
["Blooy", "ay"]

> take 5 $ iterate (1+) 0
[0, 1, 2, 3, 4]

> [1..10] \\ [1,3..10]
[2,4,6,8,10]

groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy _ [] = []
groupBy eq (x:xs) = (x:ys) : groupBy eq zs
    where (ys, zs) = span (eq x) xs

on (==) (> 0)
\x y -> (x > 0) == (y > 0)
```

- **intersperse** places a parameter inbetween items in a list.
- **intercalate** places a list between lists in a list of lists.
- **transpose** takes a list of lists and transposes them as if they were a 2D matrix.
- **foldl'** and **foldl1'** are safer versions for large lists.
- **concat** concantanates lists.
- **concatMap** first maps a function to the list and then concantanates the result.
- **and** just ANDs boolean items in a list.
- **or** just ORs boolean items in a list.
- **any** and **all** are used to simplify and/or functions by hiding the map function.
- **iterate** just creates a list using an operation and a starting item.
- **splitAt** just splits a list into two based on the given index.
- **takeWhile** takes items from a list as long as the condition is satisfied.
- **dropWhile** drops items from a list similar to takeWhile.
- **span** splits a list into two as long as a condition is satisfied.
- **breaks** breaks a list into two as soon as a condition is satisfied.
- **sort**
- **group** will create lists of adjacent identical items.
- **inits** and **tails** are like init and tail but applied recursively until the list is empty.
- **isInfixOf** will check if a subsequence is in a list.
- **isPrefixOf**
- **isSuffixof**
- **elem**
- **notElem**
- **partition** returns two lists. One that satisfies a condition and the other includes the rest.
- **find** returns the first item that satisfies a condition.
- **elemIndex** returns the index of an element.
- **elemIndices** returns returns more than one index.
- **findIndex** returns the index of the first item that satisfies the condition.
- **lines** will split a string with newlines into a list of lines.
- **unlines** does the opposite of lines.
- **words** will split a line with spaces into a list of words.

- **unwords** does the opposite of words.
- **nub** returns a list with no duplicate elements.
- **delete** deletes all instances of an item from an array.
- **backslash*2** removes items from the first parameter if they exist in the second parameter.
- **union** is like the opposite of backslash*2.
- **intersect** finds the intersection of lists.
- **insert** puts a new item in a list right before another item equal to or greater.
- Some of these functions are legacy functions replaced by more generic versions:
  - **nubBy, deleteBy, unionBy, intersectBy, groupBy**
  - Note that groupBy is a little tricky to understand so use the implementation below:
- **on** is used to simplify use on `*By` functions. The two statements below are equivalent:
- Some other new functions:
  - **sortBy, insertBy, maximumBy, and minimumBy**

Data.Char

```
> filter (isDigit) "test 12 fourty 23"
"1223"

> filter (not . isDigit) "test 12 fourty 23"
"test  fourty "

-- Caesar Cipher
> map chr $ map ((5+) . ord) "Test"
"yjxy"
```

Data.Char

- **isControl**
- **isSpace**
- **isLower**
- **isUpper**
- **isAlpha** and **isLetter** check if the character is a letter.
- **isAlphaNum** checks if the char is a letter or number.
- **isPrint** checks if char can be printed.
- **isDigit** checks 0-9.
- **isOctDigit**
- **isHexDigit**
- **isMark** checks for unicode mark chars.

- **isNumber** checks 0-9 but also roman numerals and others.
- **isPunctuation**
- **isSymbol** checks for math and currency symbols.
- **isSeparator** checks for spaces and separators.
- **isAscii**
- **isLatin1** checks if char is in the first 256 chars of Unicode.
- **isAsciiUpper**
- **isAsciiLower**
- **toUpper**
- **toLower**
- **toTitle** converts char to title-case which is usually upper-case.
- **digitToInt** converts char to int in hex.
- **intToDigit**
- **ord** converts char to their unicode value.

Data.Map

This module is for working with dictionary like structures. It works on `(key, value)` pairings.

```
> let phoneBook =
    [("a", 2),
    ("b", 4)]

findByKey :: (Eq k) => k -> [(k, v)] -> Maybe v
findByKey k [] = Nothing
findByKey k ((ks, vs):xs) = if k == ks then Just vs else findByKey k xs

findByKey2 :: (Eq k) => k -> [(k, v)] -> Maybe v
findByKey2 k = foldl (\ acc (ks, vs) -> if ks == k then Just vs else acc) Nothing
```

`Data.Map`

Note that the `foldl` has `Nothing` as it's accumulator which doesn't change but may be returned. This is a clever way to use folding.

- **fromList** takes an association list and removes duplicates and items where two keys are mapped to different values.
- **empty** returns an empty map.
- **insert** takes a key, values and a map and returns a new map with the key, values inserted.
- **null** checks for empty map.
- **size**

- **singleton** creates a new map but with a key, value you give already inserted.
- **lookup** just looks up a key, value pair and returns the value.
- **member** is like the python 'in'.
- **map**
- **filter**
- **keys** returns all the keys.
- **elems** returns all the values.
- **fromListWith** uses a function to store values instead of throwing away duplicates.
- **insertWith**

Data.Set

These are implemented using trees so they're faster than lists.

- **fromList**
- **intersection**
- **difference**
- **union**
- **size**
- **member**
- **null**
- **singleton**
- **insert**
- **delete**
- **toList** converts a set back to a list.

Custom Modules

For practice, we'll create a simple module for operations on a sphere.

```
module Sphere (
area,
volume
) where

area :: (Num a) => a -> a
area r = 4 * pi * (r ^ 2)

volume :: (Num a) => a -> a
volume r = 4 / 3 * pi * (r ^ 3)
```

Sphere Module

Importing the module can be a little annoying. Make sure that the file name is the same as the module name.

You can also have submodules. The `file.hs` should have a path like `./Geometry/Sphere.hs`. This module can be referred to as `module Geometry.Sphere`

For custom data types, you can export all the constructors by using `(..)`.

```
module Sphere (
area,
volume,
Color (..)
) where

...

data Color = Blue | Green
```

I/O

Terminal

`main` is our main like function thing that will allow us to do things that have side effects. All I/O actions and their impure actions occur inside main or a do block that is called by main.

Side effects occur when a function changes an existing value instead of returning a new value altogether.

```
main = do
    putStrLn "Name: "
    name <- getLine
    putStrLn ("Hi" ++ name)
```

Simple `main` Function

Let's look at each part of the function above.

- The `putStrLn` function takes a string and returns an I/O action. I/O actions have side effects.
- Note that the `getLine` action is "impure" because it does not guarantee the same output for the same input. We can't pass around getLine to normal functions because it is impure. We need to use it in a Monad context.

The impure functions like I/O must stay in impure environments. Note that you can never "assign" the last statement in a do block to anything. Look to the section on Monads for an explanation.

To get the result of an I/O action you must do it inside another I/O action by using `<-` . This will make more sense after understanding `monads`.

```
main = do
    putStrLn "Name?"
    raw_name <- getLine
    let name = map DC.toUpper raw_name
    putStrLn ("Hi " ++ name);

main = do
    putStrLn "Line please?"
    line <- getLine
    if null line then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

More Examples

Useful Functions

```
> mapM_ (print . (3+)) [1, 2, 3]
4
5
6
```

Examples

- **putStrLn**
- **putStr**
- **putChar**
- **print** calls show on something before putting it to terminal.
- **getChar**
- **getLine**
- **getEnv** is used to get environment variables.
- **when** useful replacement for if then. it takes a condition and if the condition is false, it ”returns” `return ()`.
- **sequence** takes a list of I/O actions and performs them one after the other.
- **mapM** and **mapM_** are used to map a function over a list but the function must return an I/O action.
- **forever** is an infinite loop like function that will perform whater I/O action you give forever.
- **forM** is like mapM but in reverse. It is usually used when you want to perform an I/O action with each element in the list.

Files and Streams

System.IO

```
main = do
    contents <- getContents
    putStrLn contents

main = interact (unlines . filter ((<10) . length) . lines)

main = interact respondPalindromes
respondPalindromes = unlines . map (\ln -> if isPalindrom ln then
    "palindrome" else "not a palindrom") . lines
    where isPalindrom str = if str == reverse str then True else False

import System.IO
main = do
    filename <- getLine
    handle <- openFile filename ReadMode
    content <- hGetContents handle
    putStr content
    hClose handle

import qualified System.IO as SIO
import qualified Data.Char as DC
main = do
    filename <- SIO.getLine
    content <- SIO.readFile filename
    SIO.writeFile "boo.txt" (map DC.toUpper content)
```

`System.IO`

- **getContents** is an I/O action that reads from the terminal but it is lazy so it waits until input is absolutely required. it's useful for piping from terminal.
- **interact** takes a `String -> String` function and then calls that on whatever it reads from the input.
- **openFile** will open a file and return an I/O Handle.
- **hGetContents** can read from that I/O Handle.
  - `hGetContents` and `getContents` are both lazy and will not do something until it is needed.
  - hGetContents also supports buffering which can be set by **hSetBuffering**.
- **hClose** will close a handle.
- **IOMode** can be ReadMode, WriteMode, AppendMode, ReadWriteMode.
- **withFile** is kind of like the `with` in python.
- **hGetLine**
- **hPutStr**
- **hPutStrLn**

- **hGetChar**
- **readFile** takes a filepath and returns IO String.
- **writeFile** takes a filepath and a string and writes it to file.
- **appendFile**
- **hFlush**
- **openTempFile**
- **removeFile**
- **renameFile**

System.Directory

- **doesFileExist** can be used instead of exceptions

Command Line

System.Environment

- **getProgName** returns a single string with the program name.
- **getArgs** returns a list of strings.

Miscellaneous

Randomness

Functions

```
Prelude System.Random> random (mkStdGen 101) :: (Int, StdGen)
(-1901866209,105509204 1655838864)
```

Randomness

- System.Random gives us some useful functions for randomness.
- **random** takes a RandomGen and a type that is a part of Random and returns a random value and another RandomGen.
- **RandomGen** is a typeclass for things that can act as sources of randomness.
- **Random** is a typeclass for things that can be random.
- **StdGen** is a type that is an instance of RandomGen.
- **mkStdGen** is a function we can use to make our own StdGen.
- **randoms** is given a StdGen and returns an infinite list of random numbers.
  - To get more random numbers we use randoms which uses the StdGen returned from the first random in the second random and so on.
- **randomR** is used to get random numbers within a range.
- **randomRs** is similar to randoms.

- **getStdGen** uses I/O to return an generator I/O action. It uses a sort of global generator.
- **newStdGen** splits our RandomGen in two and uses one of them as the new global StdGen and returns the other one as normal.

Bytestrings

The normal processing of files into strings which are lists is a little slow because of the overhead involved in making them lazy but keeping track of promises to do something.

For efficient reading and other stuff we use bytestrings. There are two types of bytestrings: strings and lazy.

- Strict
  - Strict bytestrings are in Data.ByteString.
  - These are not lazy at all.
  - They're similar to the other languages' lists.
- Lazy
  - Lazy bytestrings are a little lazier but not as much as normal lists.
  - These are processed in blocks of 64KB.

Functions

```
import qualified Data.ByteString.Lazy as DBL
import qualified Data.ByteString as DB
import qualified Data.Word as DW

> DBL.pack [23, 45]
"\ETB-"

> DBL.unpack $ DBL.pack [23, 45]
[23,45]

> DBL.unpack $ DBL.pack [0xFFFF000A, 0xF]
[10,15]
```

`Data.ByteString`

- **pack** will take a list of Word8 and turn it into a bytestring.
- **unpack** will do the reverse of pack.
  - Note how the pack will truncate a value to 8 bits.
- **fromChunks** takes a list of strict bytestrings and converts them to a lazy bytestring.
- **toChunks** takes lazy bytestring and converts it to a list of string ones.
- **cons** is like the colon in lists.
- **cons'** is the strict version.
- **empty** creates an empty bytestring.
- **readFile** and **writeFile** exist in both strict and lazy modules.

Exceptions

Haskell's type system is it's defense against failure functions. We use the `Maybe` type contructor when a pure function might fail. But we still have exceptions for some things like impure I/O for one.

catchIOError from System.IO.Error is a function we use for exceptions. It takes something to do and a handler as arguments. If the to do throws an exception it is sent to the handler.

```
;import qualified System.IO.Error as SIE
import qualified System.IO as SI
import qualified System.Environment as SE

main = SIE.catchIOError openFillet handler

openFillet :: IO ()
openFillet = do
    (filename:_) <- SE.getArgs
    hfile <- SI.openFile filename SI.ReadMode
    content <- SI.hGetContents hfile
    putStrLn "File opened successfully"
    SI.hClose hfile
    putStrLn "File closed successfully"

handler :: SIE.IOError -> IO ()
handler e
    | SIE.isDoesNotExistError e = putStrLn "File doesn't exist"
    | otherwise = SIE.ioError e
```

Exceptions

Our handler will take an `IOError` type and return an empty IO action. Inside our handler, we use functions from `System.IO.Error` to check if the errors match. these functions return a Bool value.

Note that we don't have a do block in the main because it's only one statement but other do blocks will only work if called from main.

Common IO Exceptions

- **isDoesNotExistError**
- **isAlreadyExistsError**
- **isAlreadyInUseError**
- **isFullError**
- **isEOFError**
- **isIllegalOperation**
- **isPermissionError**
- **isUserError** is thrown if we use the function userError.
- **ioe*** are a bunch of functions that return Maybe information. about the exception, e.g. **ioeGetFileName** will return a `Just filepath` or `Nothing`.

Cabal

`cabal-install` is a package installer for Haskell.

Commands

- **cabal init** will create a new cabal project in your directory.
  - Cabal will actually ask you a bunch of stuff to set up.
- **cabal configure** will configure the project according to your `.cabal` file. This file is very useful and contains everything from project name to its dependencies.
- **cabal build** will build the project for you instead of using ghc yourself.
- **cabal list** is used to search Haskell packages.
- **cabal install** is used to install a Haskell package.