Haskell Cheat Sheet

Basic syntax

Arithmetic

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
```

Boolean algebra

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

Function call

```
f x y
```

Lambdas

```
\x y -> 1 + x + y
```

Types

```
5 :: Integer
'a' :: Char
inc :: Integer -> Integer
[1, 2, 3] :: [Integer] -- equivalent to 1:(2:(3:[]))
('b', 4) :: (Char, Integer)
```

Strings are lists of characters.

Equality

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

Additional utilities

```
ghci> succ 8
9
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
ghci> div 92 10
9
ghci> mod 52 7
3
ghci> odd 3
True
```

Function definition

Functions are declared through a sequence of equations.

```
inc n = n + 1
length :: [Integer] -> Integer
length [] = 0
length (x : xs) = 1 + length xs
```

This is also an example of **pattern matching**: arguments are matched with the right parts of equations, top to bottom. If the match succeeds, the function body is called.

Parametric Polymorphism

Lower case letters are **type variables**, so `[a]` stands for a list of elements of type `a`, for any `a`

Errors

To output error messates:

```
error "Error: ..."
```

User-defined types

They are based on **data declarations**.

```
data Bool = False | True
-- a Bool can be either False or True
```

`Bool` is the **type constructor**, while `False` and `True` are **data constructors**.

```
data Pnt a = Pnt a a
-- a Pnt of type a contains two values of type a
```

Type and **data constructor** live in separate name-spaces, so it is possible (and common) to use the same name for both.

Recursive types

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

-- data constructor Branch has type:
Branch :: Tree a -> Tree a -> Tree a

-- An example tree:
aTree =
    Branch (Leaf 'a') (Branch (Leaf 'b') (Leaf 'c'))
-- in this case aTree has type Tree Char

-- An example function on trees
fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) =
    fringe left ++ fringe right
```

Lists

First of all, also lists are recursive. Indeed, they could be defined by:

```
data List a = Null | Cons a (List a)
```

However Haskell has special syntax for them; in "pseudo-Haskell":

```
data [a] = [] | a : [a]
-- [] is a data and type constructor
-- : is an infix data constructor
```

Example list:

```
numbers = [1,2,3]
```

`[1,2,3]` is actually just syntactic sugar for `1:2:3:[]`. `[]` is an empty list. If we prepend 3 to it, it becomes `[3]`. If we prepend 2 to that, it becomes `[2,3]`, and so on.

List utilities

The `++` operator appends two lists.

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
```

The `:` operator puts an element at the beginning of a list.

```
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

NB The `:` operator takes an element and a list, whereas the `++` operator takes two lists. Even if you're adding an element to the end of a list with `++`, you have to surround it with square brackets so it becomes a list.
The `!!` operator takes an element out of a list by index. The indices start at 0.

```
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

The `<`, `<=`, `>` and `>=` operators can be used to compare lists in lexicographical order. First the heads are compared. If they are equal then the second elements are compared, etc.

```
ghci> [3,4,2] > [2,4]
True
```

`head` takes a list and returns its first element.

```
ghci> head [5,4,3,2,1]
5
```

`tail` takes a list and removes its first element.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

`last` takes a list and returns its last element.

```
ghci> last [5,4,3,2,1]
1
```

`init` takes a list and returns everything except its last element.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

`length` takes a list and returns its length.

```
ghci> length [5,4,3,2,1]
5
```

`null` checks if a list is empty.

```
ghci> null [1,2,3]
False
ghci> null []
True
```

`reverse` reverses a list.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

`take` takes a number and a list. It extracts that many elements from the beginning of the list.

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

`drop` drops the number of elements from the beginning of a list.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

`maximum` returns the biggest element of a list.

```
ghci> maximum [1,9,2,3,4]
9
```

`minimum` returns the smallest.

```
minimum [8,4,2,1,5,6]
1
```

`sum` takes a list of numbers and returns their sum.

```
ghci> sum [5,2,1,6,3,2,5,7]
31
```

`product` takes a list of numbers and returns their product.

```
ghci> product [6,2,1,2]
24
```

`elem` takes a thing and a list of things and tells us if that thing is an element of the list.

```
ghci> elem 4 [3,4,5,6]
True
ghci> elem 10 [3,4,5,6]
False
```

Texas ranges

```
ghci> [1..10]
[1,2,3,4,5,6,7,8,9,10]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

Infinite lists

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
ghci> replicate 3 10
[10,10,10]
```

List comprehension

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
ghci> [x | x <- [50..100], mod x 7 == 3]
[52,59,66,73,80,87,94]
ghci> [x | x <- [10..20], x /= 13,
    x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
ghci> [x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
ghci> [ x*y | x <- [2,5,10],
    y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

Tuples

Tuples are similar to lists, but they store an exact number of heterogeneous elements. We often use tuples of two elements, called pairs, or of three elements, called triples.
E.g. `("Christopher", "Walken", 55)`.
Two useful function that operate on pairs are `fst` and `snd`. They both take a pair and return its first and second components, respectively.

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
ghci> snd (8,11)
11
ghci> snd ("Wow", False)
False
```

Another useful function is `zip`. It takes two lists and then zips them together into one list by joining the matching elements into pairs.

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
```

Syntax for fields

For *product types* (e.g. `data Point = Point Float Float`) the access is positional, for instance we may define accessors:

```
pointx Point x _ = x
pointy Point _ y = y
```

There is a C-like syntax to have **named fields**:

```
data Point = Point {pointx, pointy :: Float}
```

This declaration automatically defines two field names `pointx`, `pointy` and their corresponding selector functions.

Type synonyms

They are are defined with the keyword type, usually for readability or shortness. Examples:

```
type String = [Char]
type Assoc a b = [(a, b)]
```

Syntax in Functions

Pattern matching

The matching process proceeds top-down, left-to-right.

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

_

_ means that we really don't care what that part is.

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

x:xs

Lists themselves can also be used in pattern matching. You can match with the empty list `[]` or any pattern that involves `:` and the empty list. A pattern like `x:xs` will bind the head of the list to `x` and the rest of it to `xs`, even if there's only one element so `xs` ends up being an empty list.
NB The `x:xs` pattern is used a lot, especially with recursive functions. But patterns that have `:` in them only match against lists of length 1 or more. E.g.:

```haskell
-- Our own length implementation
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs
```

@

There's also a thing called *as patterns*. Those are a handy way of breaking something up according to a pattern and binding it to names whilst still keeping a reference to the whole thing. You do that by putting a name and an `@` in front of a pattern. For instance, the pattern `xs@(x:y:ys)`. This pattern will match exactly the same thing as `x:y:ys` but you can easily get the whole list via `xs` instead of repeating yourself by typing out `x:y:ys` in the function body again.

Boolean guards

Patterns may also have **boolean guards**.

```haskell
-- Our own max implementation
max' :: (Ord a) => a -> a -> a
max' a b
    | a > b     = a
    | otherwise = b -- optional
```

Where

Where bindings are a syntactic construct that let you bind to variables at the end of a function and the whole function can see them, including all the guards. E.g.:

```haskell
initials :: String -> String -> String
initials firstname lastname =
    [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

Let

Let bindings let you bind to variables anywhere and are expressions themselves, but are very local, so they don't span across guards. The syntax is `let <bindings> in <expression>`. The names that you define in the let part are accessible to the expression after the in part. E.g.:

```haskell
let x = 3
    y = 12
in x + y -- => 15
```

NB The difference is that `let` bindings are expressions themselves. `where` bindings are just syntactic constructs.

Case

General syntax:

```haskell
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

`expression` is matched against the patterns. The first pattern that matches the expression is used. If no suitable pattern is found, a runtime error occurs.
Example usage:

```haskell
-- take with case
take m ys = case (m,ys) of
    (0,_) -> []
    (_,[]) -> []
    (n,x:xs) -> x : take (n-1) xs
```

If

General syntax: `if <c> then <t> else <e>`.
Example:

```haskell
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

Recursion

Let's implement some functions recursively.

Maximum

```haskell
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

Replicate

```haskell
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
    | n <= 0    = []
    | otherwise = x:replicate' (n-1) x
```

Take

```haskell
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

Reverse

```haskell
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Repeat

```haskell
repeat' :: a -> [a]
repeat' x = x:repeat' x
```

Zip

```haskell
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

Elem

```haskell
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = elem' a xs
```

Quicksort

Algorithm review: a sorted list is a list that has all the values smaller than (or equal to) the head of the list in front (and those values are sorted), then comes the head of the list in the middle and then come all the values that are bigger than the head (they're also sorted).

```haskell
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Higher order functions

Map

`map` takes a function and a list and applies that function to every element in the list, producing a new list.

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

-- EXAMPLE USAGE
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
```

Filter

`filter` takes a predicate and a list and then returns the list of elements that satisfy the predicate.

```haskell
-- TYPE SIGNATURE AND DEFINITION
    filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs

-- EXAMPLE USAGE
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
```

Folds

A fold takes a binary function, a starting value (the accumulator) and a list to fold up. The binary function itself takes two parameters. The binary function is called with the accumulator and the first (or last) element and produces a new accumulator. Then, the binary function is called again with the new accumulator and the now new first (or last) element, and so on. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.

foldl

It folds the list up from the left side. The binary function is applied between the starting value and the head of the list. That produces a new accumulator value and the binary function is called with that value and the next element, etc. Definition:

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Example implementation of sum:

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

foldr

It works in a similar way to the left fold, only the accumulator eats up the values from the right. Also, the left fold's binary function has the accumulator as the first parameter and the current value as the second one (`acc x`), the right fold's binary function has the current value as the first parameter and the accumulator as the second one (`x acc`). It kind of makes sense that the right fold has the accumulator on the right, because it folds from the right side.
Definition:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Example implementation of map:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

Function composition

`.` is used for composing functions (i.e. $(f \circ g)(x) = f(g(x))$).

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Function application with $

When a $ is encountered, the expression on its right is applied as the parameter to the function on its left. It is like writing an opening parentheses and then writing a closing one on the far right side of the expression. E.g.:

```
ghci> sqrt (3 + 4 + 9)
4
ghci> sqrt $ 3 + 4 + 9
4
```

Type classes

We can declare a type to be an instance of a type class, meaning that it implements its operations. It is often not necessary to explicitly define instances of some classes, e.g. Eq and Show, as Haskell can be quite smart and do it automatically, by using `deriving`.

Class Eq

To have an instance we must implement (==). E.g.:

```
instance (Eq a) => Eq (Tree a) where
-- type a must support equality as well
Leaf a == Leaf b = a == b
(Branch l1 r1) == (Branch l2 r2) =
    (l1==l2) && (r1==r2)
_ == _ = False
```

`Eq a` is a constraint on type `a`, it means that `a` must be an instance of `Eq`.

Class Show

It is used for **showing**: to have an instance we must implement `show`. E.g.:

```
data Fpair s a = Fpair a a s | Pair a a

instance (Show a, Show s) => Show (Fpair s a) where
show (Fpair x y t) = "[" ++ (show x)
    ++ (show t) ++ (show y) ++ "]"
show (Pair x y) = "[" ++ (show x)
    ++ ", " ++ (show y) ++ "]"
```

Class Foldable

It is used for **folding**. We have a container, a binary operation `f`, and we want to apply `f` to all the elements in the container, starting from a value `z`. A minimal implementation of Foldable requires `foldr`, since `foldl` can be expressed in terms of `foldr`, while the converse is not true (since `foldr` may work on infinite lists, unlike `foldl`).
Binary tree example:

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

instance Foldable (Tree a) where
    foldr f z Empty = z
    foldr f z (Leaf x) = f x z
    foldr f z (Node l r) = foldr f (foldr f z r) l
```

Class Maybe

`Maybe` is used to represent computations that may fail: we either have `Just v`, if we are lucky, or `Nothing`.

```
data Maybe a = Nothing | Just a

-- Maybe is foldable
instance Foldable Maybe where
    foldr _ z Nothing = z
    foldr f z (Just x) = f x z
```

Class Functor

`Functor` is the class of all the types that offer a `map` operation, which is called `fmap` and has type:

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

-- Maybe is also an instance of Functor
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just a) = Just (f a)

-- Binary tree example
instance Functor (Tree a) where
    fmap f Empty = Empty
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node l r) =
        Node (fmap f l) (fmap f r)
```

Class Applicative

Applicative functors are an extended version of regular functors.
Definition:

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

`f` is a type contructor and `f a` is a Functor type. `f` must be parametric with one parameter.
If `f` is a container, the idea is the following:

- `pure` takes a value and returns the minimal `f` containing it
- `<*>` is like `fmap`, but instead of taking a simple function, it takes an `f` containing functions, applies each of them to each of the elements of a suitable container of the same kind and assembles the result in a container of again the same type

```
-- Maybe is an Applicative Functor
instance Applicative Maybe where
    pure x = Just x
    Just f <*> m = fmap f m
    Nothing <*> _ = Nothing
```

Lists are instances of Applicative

Lists are instances of Foldable and Functor. What about Applicative? Let's proceed in steps.
First, it is useful to introduce `concat`:

```
concat :: Foldable t => t [a] -> [a]
```

We start from a container of lists, and get a list with the concatenation of them.

```
ghci> concat [[1,2],[3],[4,5]]
[1,2,3,4,5]
```

It can be defined as:

```
concat l = foldr (++) [] l
```

We then define `concatMap` as its composition with `map`:

```
concatMap f l = concat (map f l)
ghci> concatMap (\x -> [x, x+1]) [1,2,3]
[1,2,2,3,3,4]
```

With `concatMap`, we get the standard implementation of `<*>` for lists:

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = concatMap (\f -> map f xs) fs
```

Or, condensing all these steps:

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = foldr (++) []
        (map (\f -> map f xs) fs)
```

What can we do with it? For instance we can apply list of operations to lists:

```
ghci> [(+1),(*2)] <*> [1,2,3]
[2,3,4,2,4,6]
```

NB we map the operations in sequence, then we concatenate the resulting lists.

Trees and Applicative

Following the list approach, we can make our binary trees an instance of Applicative Functors. First, we need to define what we mean by tree concatenation:

```
tconc Empty t = t
tconc t Empty = t
tconc t1 t2 = Node t1 t2
```

Now, `concat` and `concatMap` (here `tconcmap` for short) are like those of lists:

```
tconcat t = tfoldr tconc Empty t
tconcmap f t = tconcat (tmap f t)
```

Here is the natural definition (practically the same of lists):

```
instance Applicative Tree where
    pure x = Leaf x
    fs <*> xs = tconcmap (\f -> tmap f xs) fs
```

Again, condensed version:

```
instance Applicative Tree where
    pure x = Leaf x
    fs <*> xs = tfoldr tconc Empty
        (tmap (\f -> tmap f xs) fs)
```

General approach (for list-like types)

Let's assume our given type is `MyType`.
Step 1 - Define what is the proper concatenation (let's call it `+++`) for `MyType`, remembering that it must have type:

```
(+++) :: MyType a -> MyType a -> MyType a
```

Step 2 – Define myConcat:

```
myConcat mTp = foldr (+++) EmptyMTp mTp
```

Step 3 – Define myConcatMap:

```
myConcatMap f mTp = myConcat (fmap f mTp)
```

Step 4 – Write the definition:

```
instance Applicative (MyType a) where
pure x = ...
fs <*> xs = myConcatMap (\f -> fmap f xs) fs
```

Bonus – Compact notation:

```
instance Applicative (MyType a) where
pure x = ...
fs <*> xs = foldr (+++)
    EmptyMTp (fmap (\f -> fmap f xs) fs)
```

Class Monad

To make a type an instance of Monad, we need to implement `>>=`, called bind. The idea of `x >>= y` is that we perform the computation x, take the resulting value and pass it to y; then we perform y.
Definition:

```
(>>=) :: m a -> (a -> m b) -> m b
```

`m a` is a *computation* (or action) resulting in a value of type `a`.
Idea: The *bind* is like a function application, but instead of taking a normal value and feeding it to a normal function, it takes a monadic value (a value with a context) and feeds it to a function that takes a normal value, but returns a monadic value.
TLDR – Take your `myType a` and apply to it a function which just works on `a` and outputs a new `myType b`.

Do notation

The do syntax is used to avoid the explicit use of `>>=`. The essential translation of `do` is captured by the following:

```
e1 >>= \p -> e2
-- is translated to:

-- Version 1
do p <- e1 ; e2

-- Version 2
do p <- e1
   e2

-- Version 3
do { p <- e1 ;
     e2 }
```

Maybe

The information managed automatically by the monad is the "bit" which encodes the success (i.e. `Just`) or failure (i.e. `Nothing`) of the action sequence.

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

List

Monadic binding involves joining together a set of calculations for each value in the list. Basically, *bind* is `concatMap`.

```
instance Monad [] where
    xs >>= f = concatMap f xs
    fail _ = []
```

Trees

```
instance Monad Tree where
    xs >>= f = tconcmap f xs
    fail _ = Empty
```

The State monad

The State monad is a general monad to manage state.

Definition

First of all, let's define a type to represent our state:

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

The idea is having a type that represent a computation with a *state*, i.e. a function taking the current state and returning the next (type `a` is the explicit part of the monad).
Remember that we need unary type constructors! The "container" has now type constructor `State st`, because `State` has two parameters.

State as a functor

```
instance Functor (State st) where
    fmap f (State g) =
        State (\s -> let (s', x) = g s
                     in (s', f x))
```

The idea is quite simple: in a value of type `State st a` we apply `f` to the value of type `a`.

State as an applicative functor

```
instance Applicative (State st) where
    pure x = State (\t -> (t, x))
    (State f) <*> (State g) =
        State (\s0 -> let (s1, f') = f s0
                          (s2, x) = g s1
                      in (s2, f' x))
```

The idea is similar to the previous one: we apply `f :: State st (a -> b)` to the data part of the monad.

State as a monad

```
instance Monad (State state) where
    State f >>= g = State (\olds ->
                    let (news, value) = f olds
                        State f' = g value
                    in f' news)
```

Running the State monad

An important aspect of this monad is that monadic code does not get evaluated to data, but to a function! (Note that *State* is a function and *bind* is function composition).
In particular, we obtain a function of the *initial state*. To get a value out of it, we need to call it:

```
runStateM :: State state a -> state -> (state, a)
runStateM (State f) st = f st
```

To actually use the state, we need a way of accessing it.
The point is to move the state to the data part and back, if we want to modify it in the program.
This is easily done with these two utilities:

```
getState = State (\state -> (state, state))
putState new = State (\_ -> (new, ()))
```

Application to trees

We want to visit a tree and to give a number (e.g. a unique identifier) to each leaf.
It is of course possible to do it directly, but we need to define functions passing the current value of the id around, to be assigned and then incremented for the next leaf.
But we can also see this id as a state, and obtain we a more elegant and general definition by using our State monad.

A monadic map for trees

First we need a monadic map for trees:

```
mapTreeM f (Leaf a) = do
    b <- f a
    return (Leaf b)
mapTreeM f (Branch lhs rhs) = do
    lhs' <- mapTreeM f lhs
    rhs' <- mapTreeM f rhs
    return (Branch lhs' rhs')
```

Types

This is the type inferred by the compiler, that could work with every monad.

```
mapTreeM :: Monad m =>
    (a -> m b) -> Tree a -> m (Tree b)
```

Assigning numbers to leaves

It is now easy to do our job:

```
numberTree tree = runStateM (mapTreeM number tree) 1
    where number v = do cur <- getState
                        putState (cur+1)
                        return (v,cur)
```

Usage

Let's try it with an example tree:

```
testTree = Branch
                (Branch
                    (Leaf 'a')
                    (Branch
                        (Leaf 'b')
                        (Leaf 'c')))
                (Branch
                    (Leaf 'd')
                    (Leaf 'e'))
snd $ numberTree testTree
```

We obtain:

```
Branch (Branch  (Leaf ('a',1))
                (Branch (Leaf ('b',2))
                        (Leaf ('c',3))))
        (Branch (Leaf ('d',4)) (Leaf ('e',5)))
```

Another application: logging

In this case, instead of changing the tree, we want to implement a *logger*, that, while visiting the data structure, keeps track of the found data.
This is quite easy, if we see the log text as the state of the computation:

```
logTree tree = runStateM
    (mapTreeM collectLog tree) "Log\n"
    where collectLog v = do
        cur <- getState
        putState (cur ++ "Found node: " ++ [v] ++ "\n")
        return v
```

Usage

Let's try it with our example tree:

```
putStr $ fst $ logTree testTree
Log
Found node: a
Found node: b
Found node: c
Found node: d
Found node: e
```

Past exams

2023/09/12

#TREES

Text

Consider the *binary tree* data structure as seen in class.

1. Define a function *btrees* which takes a value x and returns an infinite list of binary trees, where:
   (a) all the leaves contain x,
   (b) each tree is complete,
   (c) the first tree is a single leaf, and each tree has one level more than its previous one in the list.
2. Define an infinite list of binary trees, which is like the previous one, but the first leaf contains the integer 1, and each subsequent tree contains leaves that have the value of the previous one incremented by one.
   E.g. *[Leaf 1, (Branch (Leaf 2)(Leaf 2), ...]*
3. Define an infinite list containing the count of nodes of the trees in the infinite list of the previous point.
   E.g. *[1, 3, ...]*

Write the signatures of all the functions you define.

Solution

```
data Btree a = Leaf a | Branch (Btree a)(Btree a)
    deriving (Show, Eq)

instance Functor Btree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch x y) = Branch (fmap f x) (fmap f y)

addLevel :: Btree a -> Btree a
addLevel t = Branch t t

btrees :: a -> [(Btree a)]
btrees x = (Leaf x) : [ addLevel t | t <- btrees x]

incBtrees :: [Btree Integer]
incBtrees = (Leaf 1) :
    [addLevel $ fmap (+1) t | t <- incBtrees]

counts :: [Integer]
counts = map (\x -> 2^x - 1) [1..]
```

2023/07/03

#LISTS

Text

1. Define a data structure, called D2L, to store lists of possibly depth two, e.g. like [1,2,[3,4],5,[6]].
2. Implement a *flatten* function which takes a D2L and returns a flat list containing all the stored values in it in the same order.
3. Make D2L an instance of Functor, Foldable, Applicative.

Solution

```
data D2L a = D2Nil | D2Cons1 a (D2L a) |
    D2Cons2 [a] (D2L a) deriving (Show, Eq)

flatten :: D2L a -> [a]
flatten D2Nil = []
flatten (D2Cons1 x ys) = x : flatten(ys)
flatten (D2Cons2 xs ys) = xs ++ flatten(ys)

instance Functor D2L where
    fmap f D2Nil = D2Nil
    fmap f (D2Cons1 x ys) =
        D2Cons1 (f x) (fmap f ys)
    fmap f (D2Cons2 xs ys) =
        D2Cons2 (fmap f xs) (fmap f ys)

instance Foldable D2L where
    foldr f i D2Nil = i
    foldr f i (D2Cons1 x ys) =
        f x (foldr f i ys)
    foldr f i (D2Cons2 xs ys) =
        foldr f (foldr f i ys) xs

(+++) :: D2L a -> D2L a -> D2L a
D2Nil +++ t = t
t +++ D2Nil = t
(D2Cons1 x ys) +++ t =
    D2Cons1 x (ys +++ t)
(D2Cons2 xs ys) +++ t =
    D2Cons2 xs (ys +++ t)

instance Applicative D2L where
    pure x = D2Cons1 x D2Nil
```

```
    fs <*> xs =
        foldr (+++) D2Nil
            (fmap (\f -> fmap f xs) fs)
```

2023/06/12

#LISTS

Text

Define a *partitioned list* data structure, called Part, storing three elements:

1. a pivot value,
2. a list of elements that are all less than or equal to the pivot, and
3. a list of all the other elements.

Implement the following utility functions, writing their types:

- *checkpart*, which takes a Part and returns true if it is valid, false otherwise;
- *part2list*, which takes a Part and returns a list of all the elements in it;
- *list2part*, which takes a pivot value and a list, and returns a Part;

Make Part an instance of Foldable and Functor, if possible. If not, explain why.

Solution

```
data Part a = Part a [a] [a] deriving (Show)

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

checkpart :: (Ord a) => Part a -> Bool
checkpart (Part p l1 l2) =
    emptyList(filter (> p) l1) &&
        emptyList(filter (<= p) l2)

-- using null instead of emptyList
checkpart2 :: Ord a => Part a -> Bool
checkpart2 (Part p l1 l2) =
    null(filter (> p) l1) && null(filter (<= p) l2)

part2list :: Part a -> [a]
part2list (Part p l1 l2) = l1 ++ [p] ++ l2

list2part :: (Ord a) => a -> [a] -> Part a
list2part p l = (Part p (filter (<= p) l)
    (filter (> p) l))

instance Foldable Part where
    foldr f z p = foldr f z (part2list p)

{-
instance Functor Part where
    fmap f (Part x a y) =
        Part (fmap f x) (f a) (fmap f y)
is not correct, because if we take e.g.
p1 = Part [1,2,3] 4 [5,6,6]; p2 = fmap (10 -) p1,
then p2 is not a correct partition.
We could use list2part to fix the solution,
but this requires that, if f :: (a -> b),
b must be an instance of Ord.
-}
instance Functor Part where
fmap :: (Ord b) => (a -> b) -> Part a -> Part b
fmap f (Part p l1 l2) = list2part p
    (fmap f (part2list (Part p l1 l2)))
```

2023/02/15

#TREES #MAYBE

Text

We want to define a data structure for binary trees, called *BBtree*, where in each node are stored two values of the same type. Write the following:

1. The *BBtree* data definition.
2. A function *bb2list* which takes a *BBtree* and returns a list with the contents of the tree.
3. Make *BBtree* an instance of Functor and Foldable.
4. Make *BBtree* an instance of Applicative, using a "zip-like" approach, i.e. every function in the first argument of `<*>` will be applied only once to the corresponding element in the second argument of `<*>`.
5. Define a function *bbmax*, together with its signature, which returns the maximum element stored in the *BBtree*, if present, or *Nothing* if the data structure is empty.

Solution

```
data BBtree t = EmptyTree |
    BBtree (BBtree t) t t (BBtree t) deriving (Eq, Show)

bb2list EmptyTree = []
bb2list (BBtree t1 a b t2) =
    (bb2list t1) ++ [a, b] ++ (bb2list t2)

instance Functor BBtree where
    fmap f (EmptyTree) = EmptyTree
    fmap f (BBtree t1 a b t2) =
        BBtree (fmap f t1) (f a) (f b) (fmap f t2)
instance Foldable BBtree where
    foldr f z EmptyTree = z
    foldr f z (BBtree t1 a b t2) =
        foldr f (f a (f b (foldr f z t2))) t1

instance Applicative BBtree where
    pure x = BBtree EmptyTree x x EmptyTree
    EmptyTree <*> _ = EmptyTree
    _ <*> EmptyTree = EmptyTree
    (BBtree t1 a b t2) <*> (BBtree t3 c d t4) =
        (BBtree (t1 <*> t3) (a c) (b d) (t2 <*> t4))

bbmax EmptyTree = Nothing
bbmax t = Just (maximum t)
-- Alternative with foldr
-- bbmax t@(BBtree t1 x y t2) =
--  Just $ foldr max x t
```

2023/01/25

#LISTS #ZIPAPPLICATIVE

Text

We want to define a data structure for the tape of a Turing machine: *Tape* is a parametric data structure with respect to the tape content, and must be made of three components:

1. the portion of the tape that is on the left of the head;
2. the symbol on which the head is positioned;
3. the portion of the tape that is on the right of the head.

Also, consider that the machine has a concept of "blank" symbols, so you need to add another component in the data definition to store the symbol used to represent the blank in the parameter type.

1. Define Tape.
2. Make Tape an instance of Show and Eq, considering that two tapes contain the same values if their stored values are the same and in the same order, regardless of the position of their heads.
3. Define the two functions *left* and *right*, to move the position of the head on the left and on the right.
4. Make Tape an instance of Functor and Applicative.

Solution

```
data Tape a = Tape [a] a [a] a
-- the last one stand for 'blank'

instance Show a => Show (Tape a) where
    show (Tape x c y b) =
        show (reverse x) ++ (show c) ++ show y
        -- reverse to conveniently implement
        -- 'left' function below

instance Eq a => Eq (Tape a) where
    (Tape x c y _) == (Tape x' c' y' _) =
        (x ++ [c] ++ y) == (x' ++ [c'] ++ y')

left :: Tape a -> Tape a
left (Tape [] c y b) = Tape [] b (c:y) b
left (Tape (x:xs) c y b) = Tape xs x (c:y) b

right (Tape x c [] b) = Tape (c:x) b [] b
right (Tape x c (y:ys) b) = Tape (c:x) y ys b

instance Functor Tape where
    fmap f (Tape x c y b) =
        Tape (fmap f x) (f c) (fmap f y) (f b)

instance Applicative Tape where
```

```haskell
    pure x = Tape [] x [] x
    (Tape fx fc fy fb) <*> (Tape x c y b) =
        Tape (zipApp fx x) (fc c)
            (zipApp fy y) (fb b) where
        zipApp x y = [f x | (f,x) <- zip x y]
```

2022/09/01

#TREES

Text

We want to implement a binary tree where in each node is stored data, together with the number of nodes contained in the subtree of which the current node is root.

1. Define the data structure.
2. Make it an instance of Functor, Foldable, and Applicative.

Solution

```haskell
data Btree t = EmptyTree |
    Btree t Int (Btree t) (Btree t) deriving (Eq, Show)

bvalue EmptyTree = 0
bvalue (Btree _ n _ _) = n

-- Auxiliary functions
bnode x t1 t2 = Btree x ((bvalue t1) +
    (bvalue t2) + 1) t1 t2
bleaf x = bnode x EmptyTree EmptyTree

instance Functor Btree where
    fmap f EmptyTree = EmptyTree
    fmap f (Btree x n t1 t2) =
        Btree (f x) n (fmap f t1) (fmap f t2)

instance Foldable Btree where
    foldr f z EmptyTree = z
    foldr f z (Btree x _ t1 t2) =
        f x (foldr f (foldr f z t2) t1)

x +++ EmptyTree = x
EmptyTree +++ x = x
(Btree x n t1 t2) +++ t = bnode x t1 (t2 +++ t)

ttconcat t = foldr (+++) EmptyTree t
ttconcmap f t = ttconcat (fmap f t)

instance Applicative Btree where
    pure x = bleaf x
    fs <*> xs = ttconcmap (\f -> fmap f xs) fs
```

2022/07/06

#QUEUES #MONAD

Text

A *deque*, short for *double-ended queue*, is a list-like data structure that supports efficient element insertion and removal from both its head and its tail. Recall that Haskell lists, however, only support O(1) insertion and removal from their head.

Implement a deque data type in Haskell by using two lists: the first one containing elements from the initial part of the list, and the second one containing elements form the final part of the list, reversed.
In this way, elements can be inserted/removed from the first list when pushing to/popping the deque's head, and from the second list when pushing to/popping the deque's tail.

1. Write a data type declaration for Deque.
2. Implement the following functions:
   - `toList`: takes a Deque and converts it to a list
   - `fromList`: takes a list and converts it to a Deque
   - `pushFront`: pushes a new element to a Deque's head
   - `popFront`: pops the first element of a Deque, returning a tuple with the popped element and the new Deque
   - `pushBack`: pushes a new element to the end of a Deque
   - `popBack`: pops the last element of a Deque, returning a tuple with the popped element and the new Deque
3. Make Deque an instance of Eq and Show.
4. Make Deque an instance of Functor, Foldable, Applicative and Monad.

You may rely on instances of the above classes for plain lists.

Solution

```haskell
data Deque a = Deque [a] [a]

toList :: Deque a -> [a]
toList (Deque front back) = front ++ reverse back

fromList :: [a] -> Deque a
fromList l = let half = div (length l) 2
    in Deque (take half l) (reverse (drop half l))

pushFront :: a -> Deque a -> Deque a
pushFront x (Deque front back) =
    Deque (x:front) back

popFront :: Deque a -> (a, Deque a)
popFront (Deque (x:front) back) =
    (x, Deque front back)
popFront (Deque [] []) = error "Pop on empty deque"
popFront (Deque [] [y]) = (y, Deque [] [])
popFront (Deque [] back) = popFront (fromList back)

pushBack :: a -> Deque a -> Deque a
pushBack x (Deque front back) = Deque front (x:back)

popBack :: Deque a -> (a, Deque a)
popBack (Deque front (x:back)) =
    (x, Deque front back)
popBack (Deque [] []) = error "Pop on empty deque"
popBack (Deque [y] []) = (y, Deque [] [])
popBack (Deque front []) = popBack (fromList front)

instance (Eq a) => Eq (Deque a) where
    d1 == d2 = toList d1 == toList d2

instance (Show a) => Show (Deque a) where
    show d = show (toList d)

instance Functor Deque where
    fmap f (Deque front back) =
        Deque (map f front) (map f back)
    -- map, since they are lists

instance Foldable Deque where
    foldr f i = foldr f i . toList -- super elegant

instance Applicative Deque where
    pure x = Deque [x] []
    fs <*> xs =
        fromList(toList fs <*> toList xs)

instance Monad Deque where
    d >>= f = fromList
        (concatMap (toList . f) (toList d))
```

2022/06/16

#PAIRS #SHOW

Text

Consider the "fancy pair" data type (called *Fpair*), which encodes a pair of the same type *a*, and may optionally have another component of some "showable" type *b*, e.g. the character '$'.
Define *Fpair*, parametric with respect to both *a* and *b*.

1. Make Fpair an instance of Show, where the implementation of show of a fancy pair e.g. encoding *(x, y, '$')* must return the string "`[x$y]`", where `x` is the string representation of *x* and `y` of *y*. If the third component is not available, the standard representation is "`[x, y]`".
2. Make *Fpair* an instance of Eq — of course the component of type *b* does not influence the actual value, being only part of the representation, so pairs with different representations could be equal.
3. Make *Fpair* an instance of Functor, Applicative and Foldable.

Solution

```haskell
-- DATA DEFINITION
data Fpair s a = Fpair a a s | Pair a a

-- SHOW INSTANCE
instance (Show a, Show s) => Show (Fpair s a) where
    show (Fpair x y t) =
        "[" ++ (show x) ++ (show t) ++ (show y) ++ "]"
```

```haskell
    show (Pair x y) =
        "[" ++ (show x) ++ ", " ++ (show y) ++ "]"

-- EQ INSTANCE (+ 'simplify' AUXILIARY FUNCTION
-- TO CONVERT TO A SIZE-TWO TUPLE (PAIR))
simplify (Fpair x y _) = (x, y)
simplify (Pair x y) = (x, y)

instance (Eq a) => Eq (Fpair s a) where
    x == y = (simplify x) == (simplify y)

-- FUNCTOR INSTANCE
instance Functor (Fpair s) where
    fmap f (Pair x y) = Pair (f x) (f y)
    fmap f (Fpair x y t) = Fpair (f x) (f y) t

-- APPLICATIVE INSTANCE
instance Applicative (Fpair s) where
    pure x = (Pair x x)
    (Pair f g) <*> (Pair x y) =
        Pair (f x) (g y)
    (Pair f g) <*> (Fpair x y t) =
        Fpair (f x) (g y) t
    (Fpair f g _) <*> (Pair x y) =
        Pair (f x) (g y)
    (Fpair f g _) <*> (Fpair x y t) =
        Fpair (f x) (g y) t

-- FOLDABLE INSTANCE
-- (since the datum is just a pair,
-- no need for recursive definition)
instance Foldable (Fpair s) where
    foldr f i (Pair x y) = f x (f y i)
    foldr f i (Fpair x y _) = f x (f y i)
```

2022/02/10

#TREES

Text

Consider a data structure Gtree for general trees, i.e. trees containg some data in each node, and a variable number of children.

1. Define the Gtree data structure.
2. Define gtree2list, i.e. a function which translates a Gtree to a list.
3. Make Gtree an instance of Functor, Foldable, and Applicative.

Solution

```haskell
data Gtree a = Tnil | Gtree a [Gtree a]
    deriving Show

-- A function which translates a Gtree to a list
gtree2list :: Gtree a -> [a]
gtree2list Tnil = []
gtree2list (Gtree x xs) =
    x : concatMap gtree2list xs

-- FUNCTOR INSTANCE
instance Functor Gtree where
    fmap f Tnil = Tnil
    fmap f (Gtree x xs) =
        Gtree (f x) (fmap (fmap f) xs)

-- FOLDABLE INSTANCE
instance Foldable Gtree where
    foldr f i t = foldr f i (gtree2list t)

-- APPLICATIVE INSTANCE
Tnil +++ x = x
x +++ Tnil = x
(Gtree x xs) +++ (Gtree y ys) =
    Gtree y ((Gtree x []:xs) ++ ys)

gtconcat = foldr (+++) Tnil

gtconcatMap f t = gtconcat (fmap f t)

instance Applicative Gtree where
    pure x = Gtree x []
    x <*> y = gtconcatMap (\f -> fmap f y) x
```

2022/01/21

#LISTS

Text

Consider a *Tvtl* (two-values/two-lists) data structure, which can store either two values of a given type, or two lists of the same type.
Define the *Tvtl* data structure, and make it an instance of Functor, Foldable, and Applicative.

Solution

```haskell
data Tvtl a = Tv a a | Tl [a] [a]
    deriving (Show, Eq)

instance Functor Tvtl where
    fmap f (Tv x y) = Tv (f x) (f y)
    fmap f (Tl x y) = Tl (fmap f x) (fmap f y)

instance Foldable Tvtl where
    foldr f i (Tv x y) = f x (f y i)
    foldr f i (Tl x y) = foldr f (foldr f i y) x

(Tv x y) +++ (Tv z w) = Tl [x, z] [y, w]
(Tv x y) +++ (Tl l r) = Tl (x:l) (y:r)
(Tl l r) +++ (Tv x y) = Tl (l ++ [x]) (r ++ [y])
(Tl l r) +++ (Tl x y) = Tl (l ++ x) (r ++ y)

tvtlconcat t = foldr (+++) (Tl [] []) t

tvtlconcmap f t = tvtlconcat (fmap f t)

instance Applicative Tvtl where
    pure x = Tl [x] []
    xs <*> ys = tvtlconcmap (\f -> fmap f ys) xs
```

Made by Antonio Sgarlata for the *Principles of Programming Languages* course held at Politecnico di Milano.
Template by Dave Richeson, Dickinson College, http://divisbyzero.com/