Recursion schemes

Or: why your abstractions are *still* weak

Koz Ross

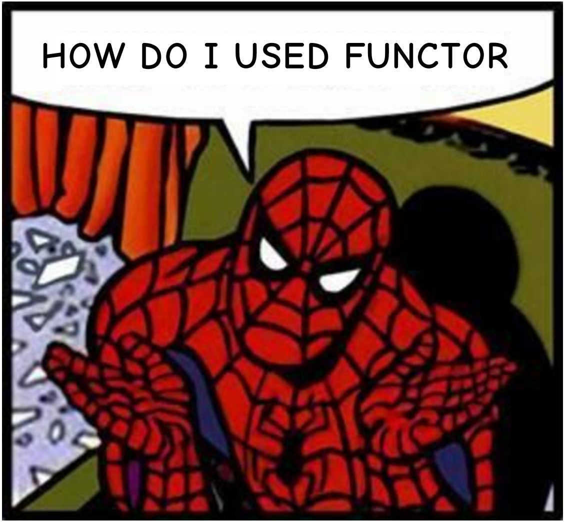
17th August, 2017

[AUT CSC]

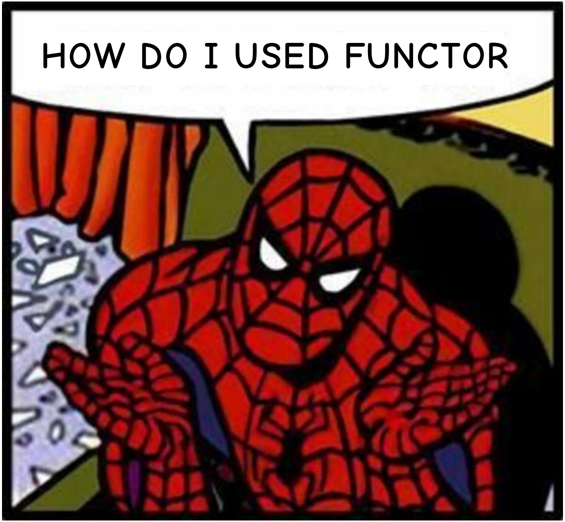
AUT Computer Science Club

Outline

How many of you felt after the functor talk



How many of you felt after the functor talk



Let me show you!

Recursive data types

We can express certain structures in a recursive manner:

Recursive data types

We can express certain structures in a recursive manner:

```
data List a =
  Nil | -- base case
  Cons a (List a) -- inductive case
```

Recursive data types

We can express certain structures in a recursive manner:

```
data List a =
  Nil | -- base case
  Cons a (List a) -- inductive case
```

Under this definition, we get the following 'expansions':

Recursive data types

We can express certain structures in a recursive manner:

```
data List a =
  Nil | -- base case
  Cons a (List a) -- inductive case
```

Under this definition, we get the following ‘expansions’:

```
[] == Nil -- obviously
[1] == Cons 1 Nil
[1, 2, 3] == Cons 1 (Cons 2 (Cons 3 Nil))
-- and so on...
```

Recursive data types

We can express certain structures in a recursive manner:

```
data List a =
  Nil | -- base case
  Cons a (List a) -- inductive case
```

Under this definition, we get the following 'expansions':

```
[] == Nil -- obviously
[1] == Cons 1 Nil
[1, 2, 3] == Cons 1 (Cons 2 (Cons 3 Nil))
-- and so on...
```

As long as we 'bottom out' somewhere, this is fine. It also allows us to write very elegant code for processing such data structures.

How to use recursive data types

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

This would lead to the following ‘expansion’:

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

This would lead to the following 'expansion':

```
-- this expression
map inc [1, 2, 3] -- Cons 1 (Cons 2 (Cons 3 Nil))
-- would be evaluated like this
```

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

This would lead to the following 'expansion':

```
-- this expression
map inc [1, 2, 3] -- Cons 1 (Cons 2 (Cons 3 Nil))
-- would be evaluated like this
Cons (inc 1) (map inc [2, 3])
```

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

This would lead to the following ‘expansion’:

```
-- this expression
map inc [1, 2, 3] -- Cons 1 (Cons 2 (Cons 3 Nil))
-- would be evaluated like this
Cons (inc 1) (map inc [2, 3])
Cons (inc 1) (Cons (inc 2) (map inc [3]))
```

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

This would lead to the following ‘expansion’:

```
-- this expression
map inc [1, 2, 3] -- Cons 1 (Cons 2 (Cons 3 Nil))
-- would be evaluated like this
Cons (inc 1) (map inc [2, 3])
Cons (inc 1) (Cons (inc 2) (map inc [3]))
Cons (inc 1) (Cons (inc 2) (Cons (inc 3) (map inc [])))
```

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

This would lead to the following 'expansion':

```
-- this expression
map inc [1, 2, 3] -- Cons 1 (Cons 2 (Cons 3 Nil))
-- would be evaluated like this
Cons (inc 1) (map inc [2, 3])
Cons (inc 1) (Cons (inc 2) (map inc [3]))
Cons (inc 1) (Cons (inc 2) (Cons (inc 3) (map inc [])))
Cons (inc 1) (Cons (inc 2) (Cons (inc 3) Nil))
```

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

This would lead to the following ‘expansion’:

```
-- this expression
map inc [1, 2, 3] -- Cons 1 (Cons 2 (Cons 3 Nil))
-- would be evaluated like this
Cons (inc 1) (map inc [2, 3])
Cons (inc 1) (Cons (inc 2) (map inc [3]))
Cons (inc 1) (Cons (inc 2) (Cons (inc 3) (map inc [])))
Cons (inc 1) (Cons (inc 2) (Cons (inc 3) Nil))
Cons (inc 1) (Cons (inc 2) (Cons 4 Nil))
```

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

This would lead to the following 'expansion':

```
-- this expression
map inc [1, 2, 3] -- Cons 1 (Cons 2 (Cons 3 Nil))
-- would be evaluated like this
Cons (inc 1) (map inc [2, 3])
Cons (inc 1) (Cons (inc 2) (map inc [3]))
Cons (inc 1) (Cons (inc 2) (Cons (inc 3) (map inc [])))
Cons (inc 1) (Cons (inc 2) (Cons (inc 3) Nil))
Cons (inc 1) (Cons (inc 2) (Cons 4 Nil))
Cons (inc 1) (Cons 3 (Cons 4 Nil))
```

How to use recursive data types

```
instance Functor List where
  map f Nil = Nil
  map f Cons head tail = Cons (f head) (map f tail)
```

This would lead to the following ‘expansion’:

```
-- this expression
map inc [1, 2, 3] -- Cons 1 (Cons 2 (Cons 3 Nil))
-- would be evaluated like this
Cons (inc 1) (map inc [2, 3])
Cons (inc 1) (Cons (inc 2) (map inc [3]))
Cons (inc 1) (Cons (inc 2) (Cons (inc 3) (map inc [])))
Cons (inc 1) (Cons (inc 2) (Cons (inc 3) Nil))
Cons (inc 1) (Cons (inc 2) (Cons 4 Nil))
Cons (inc 1) (Cons 3 (Cons 4 Nil))
Cons 2 (Cons 3 (Cons 4 Nil))-- or [2, 3, 4]
```

Going further

We can define arbitrary nested structures in this way:

Going further

We can define arbitrary nested structures in this way:

```
data FSEntry = -- file system entry
```

Going further

We can define arbitrary nested structures in this way:

```
data FSEntry = -- file system entry
  File String String | -- name, extension
```

Going further

We can define arbitrary nested structures in this way:

```
data FSEntry = -- file system entry
  File String String | -- name, extension
  Executable String | -- name
```

Going further

We can define arbitrary nested structures in this way:

```
data FSEntry = -- file system entry
  File String String | -- name, extension
  Executable String | -- name
  Folder String [FSEntry] |
```

Going further

We can define arbitrary nested structures in this way:

```
data FSEntry = -- file system entry
  File String String | -- name, extension
  Executable String | -- name
  Folder String [FSEntry] |
  Archive String String [FSEntry]
```

Going further

We can define arbitrary nested structures in this way:

```
data FSEntry = -- file system entry
  File String String | -- name, extension
  Executable String | -- name
  Folder String [FSEntry] |
  Archive String String [FSEntry]
```

Such structures are very common: trees, semi-structured data (XML, JSON, etc), expression trees, and many more, can be represented this way very naturally.

Going further

We can define arbitrary nested structures in this way:

```
data FSEntry = -- file system entry
  File String String | -- name, extension
  Executable String | -- name
  Folder String [FSEntry] |
  Archive String String [FSEntry]
```

Such structures are very common: trees, semi-structured data (XML, JSON, etc), expression trees, and many more, can be represented this way very naturally. However, working with *these* is much less easy.

Why this is awful

Let's suppose we wanted a function which told us how deeply-nested our file system is:

Why this is awful

Let's suppose we wanted a function which told us how deeply-nested our file system is:

```
depth :: FSEntry -> Integer
```

Why this is awful

Let's suppose we wanted a function which told us how deeply-nested our file system is:

```
depth :: FSEntry -> Integer
depth File _ _ = 0
```

Why this is awful

Let's suppose we wanted a function which told us how deeply-nested our file system is:

```
depth :: FSEntry -> Integer
depth File _ _ = 0
depth Executable _ = 0
```

Why this is awful

Let's suppose we wanted a function which told us how deeply-nested our file system is:

```
depth :: FSEntry -> Integer
depth File _ _ = 0
depth Executable _ = 0
depth Folder _ entries = 1 + max (map depth entries)
```

Why this is awful

Let's suppose we wanted a function which told us how deeply-nested our file system is:

```
depth :: FSEntry -> Integer
depth File _ _ = 0
depth Executable _ = 0
depth Folder _ entries = 1 + max (map depth entries)
depth Archive _ _ entries = 1 + max (map depth entries)
```

Why this is awful

Let's suppose we wanted a function which told us how deeply-nested our file system is:

```
depth :: FSEntry -> Integer
depth File _ _ = 0
depth Executable _ = 0
depth Folder _ entries = 1 + max (map depth entries)
depth Archive _ _ entries = 1 + max (map depth entries)
```

This is already looking bad. But it could be even worse...

Why this awful, part 2

Suppose we wanted a list of every file with a `.c` or `.h` extension:

Why this awful, part 2

Suppose we wanted a list of every file with a `.c` or `.h` extension:

```
findCSource :: FSEntry -> [FSEntry]
```

Why this awful, part 2

Suppose we wanted a list of every file with a `.c` or `.h` extension:

```
findCSource :: FSEntry -> [FSEntry]
findCSource File name "c" = [File name "c"]
findCSource File name "h" = [File name "h"]
```

Why this awful, part 2

Suppose we wanted a list of every file with a `.c` or `.h` extension:

```
findCSource :: FSEntry -> [FSEntry]
findCSource File name "c" = [File name "c"]
findCSource File name "h" = [File name "h"]
findCSource File _ _ = []
```

Why this awful, part 2

Suppose we wanted a list of every file with a `.c` or `.h` extension:

```
findCSource :: FSEntry -> [FSEntry]
findCSource File name "c" = [File name "c"]
findCSource File name "h" = [File name "h"]
findCSource File _ _ = []
findCSource Executable _ = []
```

Why this awful, part 2

Suppose we wanted a list of every file with a `.c` or `.h` extension:

```
findCSource :: FSEntry -> [FSEntry]
findCSource File name "c" = [File name "c"]
findCSource File name "h" = [File name "h"]
findCSource File _ _ = []
findCSource Executable _ = []
findCSource Folder _ es = concat (map findCSource es)
```

Why this awful, part 2

Suppose we wanted a list of every file with a `.c` or `.h` extension:

```
findCSource :: FSEntry -> [FSEntry]
findCSource File name "c" = [File name "c"]
findCSource File name "h" = [File name "h"]
findCSource File _ _ = []
findCSource Executable _ = []
findCSource Folder _ es = concat (map findCSource es)
findCSource Archive _ _ es = concat (map findCSource es)
```

Why this awful, part 2

Suppose we wanted a list of every file with a `.c` or `.h` extension:

```
findCSource :: FSEntry -> [FSEntry]
findCSource File name "c" = [File name "c"]
findCSource File name "h" = [File name "h"]
findCSource File _ _ = []
findCSource Executable _ = []
findCSource Folder _ es = concat (map findCSource es)
findCSource Archive _ _ es = concat (map findCSource es)
```

That is absolutely *gross*.

Why this awful, part 2

Suppose we wanted a list of every file with a `.c` or `.h` extension:

```
findCSource :: FSEntry -> [FSEntry]
findCSource File name "c" = [File name "c"]
findCSource File name "h" = [File name "h"]
findCSource File _ _ = []
findCSource Executable _ = []
findCSource Folder _ es = concat (map findCSource es)
findCSource Archive _ _ es = concat (map findCSource es)
```

That is absolutely *gross*. Now, imagine having to write 'unpack all archives, and place them in their own folders, with the same name and `"unpacked"` appended'...

Enumerating our gripes

Enumerating our gripes

- Conflates *structure* (how the data is arranged) with *content* (what the data is)

Enumerating our gripes

- Conflates *structure* (how the data is arranged) with *content* (what the data is)
- Difficult and counter-intuitive to write

Enumerating our gripes

- Conflates *structure* (how the data is arranged) with *content* (what the data is)
- Difficult and counter-intuitive to write
- Brittle and hard to extend

Enumerating our gripes

- Conflates *structure* (how the data is arranged) with *content* (what the data is)
- Difficult and counter-intuitive to write
- Brittle and hard to extend
- Tedious and repetitive

Enumerating our gripes

- Conflates *structure* (how the data is arranged) with *content* (what the data is)
- Difficult and counter-intuitive to write
- Brittle and hard to extend
- Tedious and repetitive

These are the sorts of problems ‘object-oriented’ programming should concern itself with. We, as functional programmers, should (and *can*) do better than this!

Recursion schemes

Recursion schemes

- First introduced in a 1991 paper by Meijer, Fokkinga and Paterson, titled *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*

Recursion schemes

- First introduced in a 1991 paper by Meijer, Fokkinga and Paterson, titled *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*
- Describes a set of simple functions which can arbitrarily query, tear down, and build up recursive data types

Recursion schemes

- First introduced in a 1991 paper by Meijer, Fokkinga and Paterson, titled *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*
- Describes a set of simple functions which can arbitrarily query, tear down, and build up recursive data types
- Separates structure from content, allowing easy definition and extension

Recursion schemes

- First introduced in a 1991 paper by Meijer, Fokkinga and Paterson, titled *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*
- Describes a set of simple functions which can arbitrarily query, tear down, and build up recursive data types
- Separates structure from content, allowing easy definition and extension
- Have fancy Greek names (but aren't really all that scary)

Recursion schemes

- First introduced in a 1991 paper by Meijer, Fokkinga and Paterson, titled *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*
- Describes a set of simple functions which can arbitrarily query, tear down, and build up recursive data types
- Separates structure from content, allowing easy definition and extension
- Have fancy Greek names (but aren't really all that scary)
- Not widely known (even among functional programming afficionados) or implemented (even in Haskell!)

Recursion schemes

- First introduced in a 1991 paper by Meijer, Fokkinga and Paterson, titled *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*
- Describes a set of simple functions which can arbitrarily query, tear down, and build up recursive data types
- Separates structure from content, allowing easy definition and extension
- Have fancy Greek names (but aren't really all that scary)
- Not widely known (even among functional programming afficionados) or implemented (even in Haskell!)

Meijer, Fokkinga and Paterson called these functions *morphisms*.

Recursion schemes

- First introduced in a 1991 paper by Meijer, Fokkinga and Paterson, titled *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*
- Describes a set of simple functions which can arbitrarily query, tear down, and build up recursive data types
- Separates structure from content, allowing easy definition and extension
- Have fancy Greek names (but aren't really all that scary)
- Not widely known (even among functional programming afficionados) or implemented (even in Haskell!)

Meijer, Fokkinga and Paterson called these functions *morphisms*. Let's give them a look...

First of all

To make use of recursion schemes, we have to turn our nested structure into a functor (and give it a type parameter):

First of all

To make use of recursion schemes, we have to turn our nested structure into a functor (and give it a type parameter):

```
data FSEntry a =
  File String String |
  Executable String |
  Folder String [a] |
  Archive String String [a]

instance Functor FSEntry where
  map f (File n e) = File n e
  map f (Executable n) = Executable n
  map f (Folder n es) = Folder n (map f es)
  map f (Archive n e es) = Archive n e (map f es)
```

First of all

To make use of recursion schemes, we have to turn our nested structure into a functor (and give it a type parameter):

```
data FSEntry a =
  File String String |
  Executable String |
  Folder String [a] |
  Archive String String [a]

instance Functor FSEntry where
  map f (File n e) = File n e
  map f (Executable n) = Executable n
  map f (Folder n es) = Folder n (map f es)
  map f (Archive n e es) = Archive n e (map f es)
```

The functor definition is simple enough that it can be auto-derived by the compiler. At least one compiler actually does this (GHC for Haskell).

A slight typing problem

The old `FSEntry` type was *uniform*:

A slight typing problem

The old `FSEntry` type was *uniform*:

```
file = File "foo" "txt" -- file :: FSEntry
trash = Executable "Edge" -- trash :: FSEntry
src = Folder "src" [(File "foo" "c"),
  (Executable "foo")] -- src :: FSEntry
rar = Archive "src" "rar" [src] -- rar :: FSEntry
```

A slight typing problem

The old `FSEntry` type was *uniform*:

```
file = File "foo" "txt" -- file :: FSEntry
trash = Executable "Edge" -- trash :: FSEntry
src = Folder "src" [(File "foo" "c"),
  (Executable "foo")] -- src :: FSEntry
rar = Archive "src" "rar" [src] -- rar :: FSEntry
```

The new `FSEntry`, however, is not:

A slight typing problem

The old `FSEntry` type was *uniform*:

```
file = File "foo" "txt" -- file :: FSEntry
trash = Executable "Edge" -- trash :: FSEntry
src = Folder "src" [(File "foo" "c"),
  (Executable "foo")] -- src :: FSEntry
rar = Archive "src" "rar" [src] -- rar :: FSEntry
```

The new `FSEntry`, however, is not:

```
file = File "foo" "txt" -- file :: FSEntry String
trash = Executable "Edge" -- trash :: FSEntry String
src = Folder "src" [(File "foo" "c"),
  (Executable "foo")] -- src :: FSEntry (FSEntry String)
rar = Archive "src" "rar"
  [src] -- rar :: FSEntry (FSEntry (FSEntry String))
```

A slight typing problem

The old `FSEntry` type was *uniform*:

```
file = File "foo" "txt" -- file :: FSEntry
trash = Executable "Edge" -- trash :: FSEntry
src = Folder "src" [(File "foo" "c"),
  (Executable "foo")] -- src :: FSEntry
rar = Archive "src" "rar" [src] -- rar :: FSEntry
```

The new `FSEntry`, however, is not:

```
file = File "foo" "txt" -- file :: FSEntry String
trash = Executable "Edge" -- trash :: FSEntry String
src = Folder "src" [(File "foo" "c"),
  (Executable "foo")] -- src :: FSEntry (FSEntry String)
rar = Archive "src" "rar"
  [src] -- rar :: FSEntry (FSEntry (FSEntry String))
```

This is a nuisance.

Tying the knot

We really want to have some way of saying 'this type is an arbitrarily-nested `FSEntry`', kind of like this:

Tying the knot

We really want to have some way of saying 'this type is an arbitrarily-nested `FSEntry`', kind of like this:

```
type NestedFSEntry = FSEntry (FSEntry (FSEntry …))
```

Tying the knot

We really want to have some way of saying ‘this type is an arbitrarily-nested `FSEntry`’, kind of like this:

```
type NestedFSEntry = FSEntry (FSEntry (FSEntry ...))
```

This appears impossible. However, it can be done with a bit of profitable cheating:

Tying the knot

We really want to have some way of saying ‘this type is an arbitrarily-nested `FSEntry`’, kind of like this:

```
type NestedFSEntry = FSEntry (FSEntry (FSEntry ...))
```

This appears impossible. However, it can be done with a bit of profitable cheating:

```
data Knot f = Tie { untie :: f (Knot f) }

file = Tie (File "foo" "txt") -- file :: Knot FSEntry
src = Tie (Folder "src" [(File "foo" "c"),
  (Executable "foo")]) -- src :: Knot FSEntry
rar = Tie (Archive "src" "rar"
  [src]) -- rar :: Knot FSEntry
```

Tying the knot

We really want to have some way of saying ‘this type is an arbitrarily-nested `FSEntry`’, kind of like this:

```
type NestedFSEntry = FSEntry (FSEntry (FSEntry …))
```

This appears impossible. However, it can be done with a bit of profitable cheating:

```
data Knot f = Tie { untie :: f (Knot f) }

file = Tie (File "foo" "txt") -- file :: Knot FSEntry
src = Tie (Folder "src" [(File "foo" "c"),
  (Executable "foo")]) -- src :: Knot FSEntry
rar = Tie (Archive "src" "rar"
  [src]) -- rar :: Knot FSEntry
```

This is called the *fixed point* of the `FSEntry` type, which is the same as our earlier `NestedFSEntry`.

A little helper

To make our next few steps a bit easier, we're going to define a 'pipeline' function:

A little helper

To make our next few steps a bit easier, we're going to define a 'pipeline' function:

```
-- >>> is pronounced `then'
>>> :: (a -> b) -> (b -> c) -> (a -> c)
f >>> g = g ∘ f -- ∘ is function composition
```

A little helper

To make our next few steps a bit easier, we're going to define a 'pipeline' function:

```
-- >>> is pronounced `then'
>>> :: (a -> b) -> (b -> c) -> (a -> c)
f >>> g = g ∘ f -- ∘ is function composition

f :: Integer -> Integer
f = (+ 1) >>> (* 2)
```

A little helper

To make our next few steps a bit easier, we're going to define a 'pipeline' function:

```
-- >>> is pronounced `then'
>>> :: (a -> b) -> (b -> c) -> (a -> c)
f >>> g = g ∘ f -- ∘ is function composition

f :: Integer -> Integer
f = (+ 1) >>> (* 2)

answer = f 3 -- answer == 8
```

Separating structure from content

One of the biggest annoyances of our previous scheme is all the boilerplate dealing with structure — let's be rid of it, once and for all.

Separating structure from content

One of the biggest annoyances of our previous scheme is all the boilerplate dealing with structure — let's be rid of it, once and for all. As we're dealing with tree-like structures, there's two ways we can process them:

Separating structure from content

One of the biggest annoyances of our previous scheme is all the boilerplate dealing with structure — let's be rid of it, once and for all. As we're dealing with tree-like structures, there's two ways we can process them:

Bottom-up: Descendants before ancestors

Top-down: Ancestors before descendants

Separating structure from content

One of the biggest annoyances of our previous scheme is all the boilerplate dealing with structure — let's be rid of it, once and for all. As we're dealing with tree-like structures, there's two ways we can process them:

Bottom-up: Descendants before ancestors

Top-down: Ancestors before descendants

This is where our `Functor` instance can really shine — we can use it to automate the 'process descendants' step, without worrying about our structure or breaking it (due to the functor law).

Separating structure from content

One of the biggest annoyances of our previous scheme is all the boilerplate dealing with structure — let's be rid of it, once and for all. As we're dealing with tree-like structures, there's two ways we can process them:

Bottom-up: Descendants before ancestors

Top-down: Ancestors before descendants

This is where our `Functor` instance can really shine — we can use it to automate the 'process descendants' step, without worrying about our structure or breaking it (due to the functor law). We just need to provide a *local* processing function, which deals with our content.

Bottom-up processing

To process a recursive data structure *bottom-up* using the processing function `f`, we need to take the following steps:

Bottom-up processing

To process a recursive data structure *bottom-up* using the processing function `f`, we need to take the following steps:

1. Untie the `Knot`

Bottom-up processing

To process a recursive data structure *bottom-up* using the processing function `f`, we need to take the following steps:

1. Untie the `Knot`
2. Process descendants using `map f`

Bottom-up processing

To process a recursive data structure *bottom-up* using the processing function `f`, we need to take the following steps:

1. Untie the `Knot`
2. Process descendants using `map f`
3. Re-tie the `Knot`

Bottom-up processing

To process a recursive data structure *bottom-up* using the processing function `f`, we need to take the following steps:

1. Untie the `Knot`
2. Process descendants using `map f`
3. Re-tie the `Knot`
4. Process the ancestor using `f`

Bottom-up processing

To process a recursive data structure *bottom-up* using the processing function `f`, we need to take the following steps:

1. Untie the `Knot`
2. Process descendants using `map f`
3. Re-tie the `Knot`
4. Process the ancestor using `f`

In code terms:

Bottom-up processing

To process a recursive data structure *bottom-up* using the processing function `f`, we need to take the following steps:

1. Untie the `Knot`
2. Process descendants using `map f`
3. Re-tie the `Knot`
4. Process the ancestor using `f`

In code terms:

```
bottomUp :: Functor a => (Knot a -> Knot a) -> Knot a -> Knot a
bottomUp f = untie >>> map (bottomUp f) >>> Tie >>> f
```

Bottom-up processing

To process a recursive data structure *bottom-up* using the processing function `f`, we need to take the following steps:

1. Untie the `Knot`
2. Process descendants using `map f`
3. Re-tie the `Knot`
4. Process the ancestor using `f`

In code terms:

```
bottomUp :: Functor a => (Knot a -> Knot a) -> Knot a -> Knot a
bottomUp f = untie >>> map (bottomUp f) >>> Tie >>> f
```

Note: `bottomUp` absolutely and totally does not care what we tied into a `Knot`. Furthermore, `f` doesn't need to care about the structure anymore.

Top-down processing

To process a recursive data structure *top-down* using the processing function `f`, we need to take the following steps:

Top-down processing

To process a recursive data structure *top-down* using the processing function `f`, we need to take the following steps:

1. Process the ancestor using `f`

Top-down processing

To process a recursive data structure *top-down* using the processing function `f`, we need to take the following steps:

1. Process the ancestor using `f`
2. Untie the `Knot`

Top-down processing

To process a recursive data structure *top-down* using the processing function `f`, we need to take the following steps:

1. Process the ancestor using `f`
2. Untie the `Knot`
3. Process descendants using `map f`

Top-down processing

To process a recursive data structure *top-down* using the processing function `f`, we need to take the following steps:

1. Process the ancestor using `f`
2. Untie the `Knot`
3. Process descendants using `map f`
4. Re-tie the `Knot`

Top-down processing

To process a recursive data structure *top-down* using the processing function `f`, we need to take the following steps:

1. Process the ancestor using `f`
2. Untie the `Knot`
3. Process descendants using `map f`
4. Re-tie the `Knot`

In code terms:

Top-down processing

To process a recursive data structure *top-down* using the processing function `f`, we need to take the following steps:

1. Process the ancestor using `f`
2. Untie the `Knot`
3. Process descendants using `map f`
4. Re-tie the `Knot`

In code terms:

```
topDown :: Functor a => (Knot a -> Knot a) -> Knot a -> Knot a
topDown f = f >>> untie >>> map (topDown f) >>> Tie
```

Top-down processing

To process a recursive data structure *top-down* using the processing function `f`, we need to take the following steps:

1. Process the ancestor using `f`
2. Untie the `Knot`
3. Process descendants using `map f`
4. Re-tie the `Knot`

In code terms:

```
topDown :: Functor a => (Knot a -> Knot a) -> Knot a -> Knot a
topDown f = f >>> untie >>> map (topDown f) >>> Tie
```

Note: `topDown` absolutely and totally does not care what we tied into a `Knot`. Furthermore, `f` doesn't need to care about the structure anymore.

Slipping the `Knot`

Unfortunately, our separation of structure and content is not complete — we're still tied (hurr hurr) to returning a `Knot`, which is not enough to implement anything (too) useful.

Slipping the `Knot`

Unfortunately, our separation of structure and content is not complete — we're still tied (hurr hurr) to returning a `Knot`, which is not enough to implement anything (too) useful. What if we tried to 'slip the `Knot`' by 'cleverly' forgetting to re-tie it?

Slipping the `Knot`

Unfortunately, our separation of structure and content is not complete — we're still tied (hurr hurr) to returning a `Knot`, which is not enough to implement anything (too) useful. What if we tried to 'slip the `Knot`' by 'cleverly' forgetting to re-tie it?

```
-- what would the type of this possibly be?
mystery f = untie >>> map (mystery f) >>> f
```

Slipping the `Knot`

Unfortunately, our separation of structure and content is not complete — we're still tied (hurr hurr) to returning a `Knot`, which is not enough to implement anything (too) useful. What if we tried to 'slip the `Knot`' by 'cleverly' forgetting to re-tie it?

```
-- what would the type of this possibly be?
mystery f = untie >>> map (mystery f) >>> f

-- mystery :: Functor f => (f a -> a) -> Knot f -> a
```

Slipping the Knot

Unfortunately, our separation of structure and content is not complete — we're still tied (hurr hurr) to returning a `Knot`, which is not enough to implement anything (too) useful. What if we tried to 'slip the `Knot`' by 'cleverly' forgetting to re-tie it?

```
-- what would the type of this possibly be?
mystery f = untie >>> map (mystery f) >>> f

-- mystery :: Functor f => (f a -> a) -> Knot f -> a
```

Now, the separation is complete. Let's write some code!

Revisiting `depth`

Revisiting depth

```
countDepth :: FSEntry Int -> Int
countDepth File _ _ = 0
countDepth Executable _ = 0
countDepth Folder _ es = 1 + max es
countDepth Executable _ es = 1 + max es
```

Revisiting depth

```
countDepth :: FSEntry Int -> Int
countDepth File _ _ = 0
countDepth Executable _ = 0
countDepth Folder _ es = 1 + max es
countDepth Executable _ es = 1 + max es

depth :: Knot FSEntry -> Int
depth = mystery countDepth
```

Revisiting depth

```
countDepth :: FSEntry Int -> Int
countDepth File _ _ = 0
countDepth Executable _ = 0
countDepth Folder _ es = 1 + max es
countDepth Executable _ es = 1 + max es

depth :: Knot FSEntry -> Int
depth = mystery countDepth
```

`countDepth` doesn't have to care about structure, while `mystery` doesn't have to care about content. Success!

Revisiting `findCSource`

Revisiting `findCSource`

```
getCSource :: FSEntry String -> [FSEntry String]
getCSource File n "c" = [File name "c"]
getCSource File n "h" = [File name "h"]
getCSource File _ _  = []
getCSource Executable _ = []
getCSource Folder _ es = concat es
getCSource Archive _ _ es = concat es
```

Revisiting `findCSource`

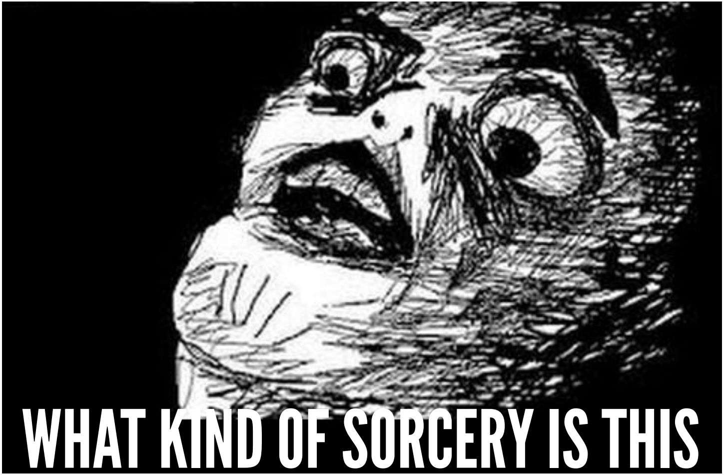
```
getCSource :: FSEntry String -> [FSEntry String]
getCSource File n "c" = [File name "c"]
getCSource File n "h" = [File name "h"]
getCSource File _ _  = []
getCSource Executable _ = []
getCSource Folder _ es = concat es
getCSource Archive _ _ es = concat es

findCSource :: Knot FSEntry -> [FSEntry]
findCSource = mystery getCSource
```

Revisiting `findCSource`

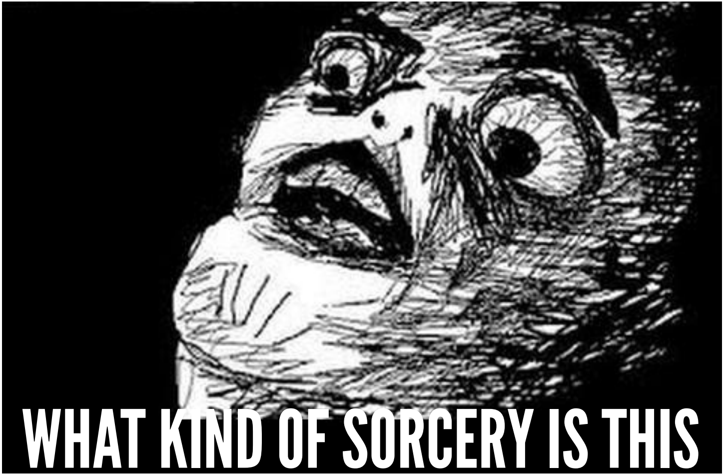
```
getCSource :: FSEntry String -> [FSEntry String]
getCSource File n "c" = [File name "c"]
getCSource File n "h" = [File name "h"]
getCSource File _ _  = []
getCSource Executable _ = []
getCSource Folder _ es = concat es
getCSource Archive _ _ es = concat es

findCSource :: Knot FSEntry -> [FSEntry]
findCSource = mystery getCSource
```

Woo-hoo!

What you might be thinking right now



What you might be thinking right now



I assure you, it gets better!

Cleaning up our terminology

`mystery` is a pretty silly name to give something so useful, and `(f a -> a)` is really hard to pronounce. Let's name them sensibly:

Cleaning up our terminology

`mystery` is a pretty silly name to give something so useful, and `(f a -> a)` is really hard to pronounce. Let's name them sensibly:

```
type Algebra f a = f a -> a

-- the humble catamorphism, at last
cata :: Functor f => Algebra f a -> Knot f -> a
cata f = untie >>> map (cata f) >>> f
```

Cleaning up our terminology

`mystery` is a pretty silly name to give something so useful, and `(f a -> a)` is really hard to pronounce. Let's name them sensibly:

```
type Algebra f a = f a -> a

-- the humble catamorphism, at last
cata :: Functor f => Algebra f a -> Knot f -> a
cata f = untie >>> map (cata f) >>> f
```

The names seem a bit odd at first, but there are reasons for them:

Cleaning up our terminology

`mystery` is a pretty silly name to give something so useful, and `(f a -> a)` is really hard to pronounce. Let's name them sensibly:

```
type Algebra f a = f a -> a

-- the humble catamorphism, at last
cata :: Functor f => Algebra f a -> Knot f -> a
cata f = untie >>> map (cata f) >>> f
```

The names seem a bit odd at first, but there are reasons for them:

Algebra: Arabic root *jabr*, which means 'restoration, reunion' — an *algebra* 'reunites' an `f a` (a container of `a`s) back into a single `a`.

Cleaning up our terminology

`mystery` is a pretty silly name to give something so useful, and `(f a -> a)` is really hard to pronounce. Let's name them sensibly:

```
type Algebra f a = f a -> a

-- the humble catamorphism, at last
cata :: Functor f => Algebra f a -> Knot f -> a
cata f = untie >>> map (cata f) >>> f
```

The names seem a bit odd at first, but there are reasons for them:

Algebra: Arabic root *jabr*, which means 'restoration, reunion' — an *algebra* 'reunites' an `f a` (a container of `a`s) back into a single `a`.

Catamorphism: Greek root *kata* (like 'catastrophe'), which means 'downwards, into, collapse' — a *catamorphism* 'collapses' a nested structure of values into a single value.

Extending to logical conclusions

We just managed to decouple structure from content for *bottom-up* traversals. But what if we tried to do the same for top-down ones?

Extending to logical conclusions

We just managed to decouple structure from content for *bottom-up* traversals. But what if we tried to do the same for top-down ones?

```
-- a reminder
topDown f = f >>> untie >>> map (topDown f) >>> Tie
```

Extending to logical conclusions

We just managed to decouple structure from content for *bottom-up* traversals. But what if we tried to do the same for top-down ones?

```
-- a reminder
topDown f = f >>> untie >>> map (topDown f) >>> Tie

-- what is the type of this?
wtf f = f >>> map (wtf f) >>> Tie
```

Extending to logical conclusions

We just managed to decouple structure from content for *bottom-up* traversals. But what if we tried to do the same for top-down ones?

```
-- a reminder
topDown f = f >>> untie >>> map (topDown f) >>> Tie

-- what is the type of this?
wtf f = f >>> map (wtf f) >>> Tie
-- wtf :: (Functor f) => (a -> f a) -> a -> Knot f
```

Extending to logical conclusions

We just managed to decouple structure from content for *bottom-up* traversals. But what if we tried to do the same for top-down ones?

```
-- a reminder
topDown f = f >>> untie >>> map (topDown f) >>> Tie

-- what is the type of this?
wtf f = f >>> map (wtf f) >>> Tie
-- wtf :: (Functor f) => (a -> f a) -> a -> Knot f
```

Seem familiar?

Wtf is up with `wtf`?

Consider the type signatures of `cata` versus `wtf`:

Wtf is up with `wtf`?

Consider the type signatures of `cata` versus `wtf`:

`cata`: Takes an algebra and a nested structure, and 'tears it down' into a single value

Wtf is up with `wtf`?

Consider the type signatures of `cata` versus `wtf`:

- `cata`: Takes an algebra and a nested structure, and 'tears it down' into a single value
- `wtf`: Takes a 'backwards-algebra' and a value, and 'builds it up' into a nested structure

Wtf is up with `wtf`?

Consider the type signatures of `cata` versus `wtf`:

`cata`: Takes an algebra and a nested structure, and 'tears it down' into a single value

`wtf`: Takes a 'backwards-algebra' and a value, and 'builds it up' into a nested structure

Again, sensible naming is in order:

Wtf is up with `wtf`?

Consider the type signatures of `cata` versus `wtf`:

- `cata`: Takes an algebra and a nested structure, and 'tears it down' into a single value
- `wtf`: Takes a 'backwards-algebra' and a value, and 'builds it up' into a nested structure

Again, sensible naming is in order:

```
type Coalgebra f a = a -> f a

-- meet the anamorphism (from Greek root for `build')
ana :: Functor f => Coalgebra f a -> a -> Knot f
ana f = f >>> map (ana f) >>> Tie
```

Wtf is up with `wtf`?

Consider the type signatures of `cata` versus `wtf`:

- `cata`: Takes an algebra and a nested structure, and 'tears it down' into a single value
- `wtf`: Takes a 'backwards-algebra' and a value, and 'builds it up' into a nested structure

Again, sensible naming is in order:

```
type Coalgebra f a = a -> f a

-- meet the anamorphism (from Greek root for `build')
ana :: Functor f => Coalgebra f a -> a -> Knot f
ana f = f >>> map (ana f) >>> Tie
```

Thanks to recursion schemes, we can now *build* nested structures up based on local expansions.

Why recursion schemes are amazing

Why recursion schemes are amazing

- Full separation of structure and content

Why recursion schemes are amazing

- Full separation of structure and content
- Easy to modify, simpler to write

Why recursion schemes are amazing

- Full separation of structure and content
- Easy to modify, simpler to write
- Formal guarantees of correct behaviour

Why recursion schemes are amazing

- Full separation of structure and content
- Easy to modify, simpler to write
- Formal guarantees of correct behaviour

All this is possible because of the wonderful `Functor`!

Going further

Going further

- Context-sensitive construction and teardown (*paramorphisms* and *apomorphisms*)

Going further

- Context-sensitive construction and teardown (*paramorphisms* and *apomorphisms*)
- Processing via intermediate structures, generalization and simplification of divide-and-conquer (*hylomorphisms*)

Going further

- Context-sensitive construction and teardown (*paramorphisms* and *apomorphisms*)
- Processing via intermediate structures, generalization and simplification of divide-and-conquer (*hylomorphisms*)
- Generalize more expressive techniques (for example, trampolining with *zygomorphisms* and dynamic programming with *histomorphisms*)

Going further

- Context-sensitive construction and teardown (*paramorphisms* and *apomorphisms*)
- Processing via intermediate structures, generalization and simplification of divide-and-conquer (*hylomorphisms*)
- Generalize more expressive techniques (for example, trampolining with *zygomorphisms* and dynamic programming with *histomorphisms*)
- Processing *infinite* structures (yes, this is possible!)

Going further

- Context-sensitive construction and teardown (*paramorphisms* and *apomorphisms*)
- Processing via intermediate structures, generalization and simplification of divide-and-conquer (*hylomorphisms*)
- Generalize more expressive techniques (for example, trampolining with *zygomorphisms* and dynamic programming with *histomorphisms*)
- Processing *infinite* structures (yes, this is possible!)

But ultimately, this is *freedom from boilerplate*, *freedom from drudgery*, and ultimately, *freedom from structural processing*.

Going further

- Context-sensitive construction and teardown (*paramorphisms* and *apomorphisms*)
- Processing via intermediate structures, generalization and simplification of divide-and-conquer (*hylomorphisms*)
- Generalize more expressive techniques (for example, trampolining with *zygomorphisms* and dynamic programming with *histomorphisms*)
- Processing *infinite* structures (yes, this is possible!)

But ultimately, this is *freedom from boilerplate*, *freedom from drudgery*, and ultimately, *freedom from structural processing*.

Still don't know how to used functor?

Questions?