

Data Origami

Austin Haskell Meetup

August 18, 2016

Questions from last time?

Recursion

- ▶ Usually hear “functions that call themselves”

Other ways to think about it

- ▶ “solving a problem in terms of smaller versions of the same problem” (John D. Cook)
- ▶ computations that have to be performed an *indefinite* number of times

Ye Olde Factorial

```
brokenFact1 :: Integer -> Integer  
brokenFact1 n = n * brokenFact1 (n - 1)
```

- ▶ How will that evaluate?
- ▶ Try it in REPL

Evaluation

```
brokenFact1 4 = 4 * (4 - 1)
               * ((4 - 1) - 1)
               * (((4 - 1) - 1) - 1)
               ... this series never stops
```

- Well, that seems suboptimal.

Ye Olde Base Case

```
factorial :: Integer -> Integer
factorial 1 = 1
factorial n = n * brokenFact1 (n - 1)
```

- ▶ if the base case is an identity value, doesn't change the results of previous applications

A Recursive Datatype

```
data [] a = [] | a : [a]
```

- ▶ cons constructor : is an infix data constructor
- ▶ a product of its two arguments
- ▶ a potentially infinite data stream!

Pattern matching on lists

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

```
tail :: [a] -> [a]  
tail (x:xs) = xs
```

- ▶ use `let` in REPL
- ▶ try to pass them empty lists

Rewriting for fun and safety

- ▶ an empty list
- ▶ Maybe?
- ▶ Either?

```
myHead :: [a] -> Either [Char] a
myHead [] = Left "Empty list."
myHead (x:_) = Right x
```

Either

```
data Either a b = Left a | Right b
```

- ▶ like Maybe often used to prevent bottoming out
- ▶ provides opportunity here to tell *what* the error was

Exercise 1

- ▶ recommend doing this in a source file instead of directly in REPL

Seeing evaluation using :sprint

- ▶ `enumFromTo` *constructs* a list

```
Prelude> let blah = enumFromTo 'a' 'z'
```

```
Prelude> :sprint blah
```

```
blah = _
```

```
Prelude> take 1 blah
```

```
"a"
```

- ▶ normally doesn't evaluate until forced

Spine strictness

- ▶ matters when we talk about folds, binary trees
- ▶ evaluates to weak head normal form by default

Compare

- ▶ see which of these (if any) throws an exception

```
[x^y | x <- [1..5], y <- [2, undefined]]
```

```
take 1 $ [x^y | x <- [1..5], y <- [2, undefined]]
```

Time for a map

`map :: (a -> b) -> [a] -> [b]`

- ▶ obligatory reminder that data structures are (usually) immutable in Haskell :)
- ▶ write `map` (exercise 2)

And filter

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

- ▶ hey, how do we know there's no mutation here?
- ▶ try Exercise 3
- ▶ why would you need words for this?

Exercise 3 answer

```
noDets :: String -> [[Char]]
noDets =
  filter (\x -> not (elem x ["the", "a", "an"])) . words
```

Exercises

- ▶ will rewrite with folds later

-- direct recursion, not using (fold)

```
myAnd :: [Bool] -> Bool
```

```
myAnd [] = True
```

```
myAnd (x:xs) = if x == False then False else myAnd xs
```

- ▶ see exercise 4

Answers

```
myOr' :: [Bool] -> Bool
myOr' [] = False
myOr' (x:xs)
  | x = x
  | otherwise = myOr' xs
```

```
myAny' :: (a -> Bool) -> [a] -> Bool
myAny' _ [] = False
myAny' f (x:xs)
  | f x = True
  | otherwise = myAny' f xs
```