Exercises 3.3 and 3.4

F142001

June 19, 2015

Some preliminaries:

```
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE FlexibleContexts #-}

import Generics.Regular
```

Exercise 3

First, the definition of the `Children` class (with all of their `Regular` instances) and the `children` functions. They are pretty straightforward: only at the `I` constructor do we find any children. Since we are talking about children, and not grandchildren etc, we don't even need recursion.

```
class Children (p :: * -> *) where
  children' :: p r -> [r]

children :: (Regular r, Children (PF r)) => r -> [r]
children = children' . from

instance Children U where
  children' U = []

instance Children (K a) where
  children' (K x) = []

instance (Children a, Children b) => Children (a :+: b) where
  children' (L x) = children' x
  children' (R x) = children' x

instance (Children a, Children b) => Children (a :*: b) where
  children' (x :*: y) = (children' x) ++ (children' y)

instance Children I where
  children' (I x) = x : []
```

```
instance (Children a) => Children (C c a) where
  children' (C x) = children' x
```

An example to test with: lists. Any list has (at most) one child.

```
type instance PF [a] = U :+: K a :*: I

instance Regular [a] where
  from [] = L U
  from (x : xs) = R ( (K x) :*: (I xs))
  to (L U) = []
  to (R (K x :*: (I xs))) = x : xs

exampleList = children [1,2,3]
```

Another example: binary trees. Every tree has either 0 or 2 recursive children.

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

type instance PF (Tree a) = K a :+: (I :*: I)

instance Regular (Tree a) where
  from (Leaf x) = L (K x)
  from (Bin t1 t2) = R ((I t1) :*: (I t2))
  to (L (K x)) = Leaf x
  to (R ((I t1) :*: (I t2))) = Bin t1 t2

instance (Show a) => Show (Tree a) where
  show (Leaf x) = "(Leaf " ++ show x ++ ")"
  show (Bin t1 t2) = "(Bin " ++ show t1 ++ " " ++ show t2 ++ ")"

exampleTree = children (Bin (Bin (Leaf 1) (Leaf 2)) (Leaf 3))
```

Exercise 4

This is a short one: if an expression does not have *any* children, it is itself not a parent, and of course, none of its children are (because it doesn't have any). If an expression does have children, it is a parent, and we recursively check whether its children are parents (and their children, etc)

```
parents :: (Regular r, Children (PF r)) => r -> [r]
parents x = case children x of
  [] -> []
  (c : cs) -> x : concat (map parents (c : cs))

exampleParents = parents [1,2,3]
```