Problem Set #6

due October 25

Problem 1. In this problem you will draw parse trees of lambda terms as we did in class. This is practice with Lambda calculus syntax, to help with understanding of constraint generation for type inference (which is syntax-based).

(1) $\lambda f.\lambda x.\ f\ (f\ x)$

(2) $(\lambda x.\lambda y.\lambda z.x\ z\ (y\ z))\ (\lambda u.u)(\lambda v.v)$

(3) $(\lambda x.\lambda y.x\ y)\ (\lambda z.(\lambda x.\lambda y.x)\ z\ ((\lambda x.z\ x)(\lambda x.z\ x)))$

You can draw by hand and scan your solution (though it is better if you typed in latex).

Problem 2. Next, we'll practice lazy evaluation. First, define a polymorphic data type `InfList` representing *infinite* lists. `InfList`s are like lists, except that they have only a "Cons" constructor. In contrast, regular lists have a "Cons" and an "Empty" constructors.

(1) Write a function to convert an infinite list to a regular list.

```
infToList :: InfList a -> [a]
infToList ...

> take 5 $ infToList $ infRepeat 1
[1,1,1,1,1]
```

(2) Write a function to generate an infinite list of some element.

```
infRepeat :: a -> InfList a
infRepeat ...

> infRepeat 1
[1,1,1,1,1,1,1,1,1,1] -- assuming show prints first 10 elements
```

(3) Make your list an instance of `Show`, so that one can show (first n elements of) an infinite list in the read-eval-print loop.

```
instance (Show a) => Show (InfList a) where
...
```

(4) Write a function to map an infinite list of `a`'s into an infinite list of `b`'s.

```
infMap :: (a -> b) -> InfList a -> InfList b
infMap ...

> infMap (+1) $ infRepeat 1
[2,2,2,2,2,2,2,2,2,2] -- assuming show prints first 10 elements
```

(5) Write a function to generate an infinite list from a seed (essentially `iterate` over infinite lists).

```
infFromSeed :: (a -> a) -> a -> InfList a
infFromSeed ...

> infFromSeed (+10) 0
[0,10,20,30,40,50,60,70,80,90] -- assuming show prints first 10 elements
```

(6) Define the infinite list `nats` of natural numbers 1,2,3,..

```
nats :: InfList Integer
nats ...
```

(7) Define the infinite list `ruler` such that the n-th element (starting at $n = 1$) is the highest power of 2 that divides n.

```
ruler :: InfList Integer
ruler ...

> ruler
[0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,4,0,1,0,2] -- assuming show prints first 20
```

For full credit, write a helper that *interleaves* two infinite lists and a single short line for `ruler`.

Note: As always, write your code in file `Ps6.hs` and submit in Submitty. For grading on Submitty, set the number of elements `show` prints to 20.

```
{-# OPTIONS_GHC -Wall #-}

module Ps6 where
...
```

Haskell Style Guide. Adapted from Stephanie Weirich (UPenn CIS 5520).

- **Write a type signature for every function.** We will be strict about this when grading. Hint: try writing the signature *before* writing the function. If you do write the function first, try deducing the signature and if this doesn't work, there is always the `:t` command.
- Make sure that your code produces no errors or warnings. Code with errors receives 0 on Submitty and we will mark down warnings during TA grading.

- Use consistent indentation.
- Do not use tab characters, use space for indentation. GHC should be flagging tabs, but nevertheless, be careful.
- No line should have more than **80 characters**.
- Use whitespace to make your code readable. Add whitespace on either side of binary operators, e.g., write `3 * n + 1` instead of `3*n+1`.

- Use descriptive names.
- Follow standard Haskell naming conversions: (1) use camelCase for compound names, and (2) use `x` and `xs` when you pattern-match lists.

- Use comments. Each function definition should be preceded by a comment.
- Comment should say what the function does, not how.
- Comments should be concise. Do not overcomment.
- Use full English sentences.

- Do not leave incomplete pattern matches. They will be marked down.
- Tuples, records and datatypes can be decomposed. You can also use the `@` operator if you need a reference to both the object and its components.
  For example, do not use this:

```
f arg1 arg2 = ...  where
   x = fst arg1
   y = snd arg1
   z = fst arg2
```

  Use this instead:

```
f (x,y) (z,_) = ...
```

- Combine nested case expressions.
  For example, do not use this:

```
case x of
      Red -> case y of
                Red  -> True
                Blue -> False
      Blue -> case y of
                Red  -> False
                Blue -> True
```

  Use this instead:

```
case (x,y) of
```

```
(Red,   Red) -> True
(Blue, Blue) -> True
(   _,    _) -> False
```

- Use library functions, unless the assignment explicitly forbids them. Use Haskell's search engine Hoogle to look up library functions.