Assignment 2 (Due: Weds Sep 15 at 5pm) Computational Linguistics (LING 455)

Directions: This assignment has **6** questions. There are two files: `HW2.hs`, which is where you should put your answers, and the easier to read `HW2.pdf`. Your answers will either be replacing `undefined` pieces of code, or tracing evaluation steps (the second of these must be done in a comment, since these evaluation steps aren't themselves valid Haskell). Start by loading `HW2.hs` into `ghci`, and periodically check your work by saving your changes and using `:r` to refresh/update `ghci` with them.

1 Warm up

We'll start by defining a type `Shape` for the possible moves in Rock, Paper, Scissors, and a type `Result` for the possible outcomes of one game.

```
data Shape = Rock | Paper | Scissors
  deriving Show
-- The second line of this definition and the next, which you can ignore, just
-- says we'd like these values to display on the screen in the obvious way.

data Result = Win | Lose | Tie
  deriving Show
```

(1) `whatItBeats` is a function that takes a `Shape` as an input and returns the shape that it beats. `play` is a function taking two `Shape`s, which tells if the first `Shape` won, lost, or tied. These definitions are incomplete. Complete them by replacing each occurrence of `undefined` with the right value.

```
whatItBeats :: Shape -> Shape
whatItBeats Rock     = Scissors
whatItBeats Paper    = undefined
whatItBeats Scissors = undefined

play :: Shape -> Shape -> Result
play Rock     Paper    = Lose
play Rock     Scissors = undefined
play Paper    Rock     = undefined
play Paper    Scissors = undefined
play Scissors Rock     = undefined
play Scissors Paper    = undefined
play _        _        = Tie -- All other possibilities give a Tie
```

2 Practice with evaluation

(2) Using the above definitions and the evaluation rules for `let` and lambda, evaluate the following three expressions, one step at a time, by filling out each of the blank lines below. You can check your answers by pasting the starting expression into `ghci` and having it evaluate it for you! (But you will be graded on the intermediate steps too!)

a. `let guaranteedTie = (\x -> play x x) in guaranteedTie Paper`

$\Longrightarrow$

$\Longrightarrow$

$\Longrightarrow$

b. `let unbeatable = (\y -> play y (whatItBeats y)) in unbeatable Scissors`
$\Longrightarrow$
$\Longrightarrow$
$\Longrightarrow$
$\Longrightarrow$

c. `(\f -> f (f (f Rock))) whatItBeats`
$\Longrightarrow$
$\Longrightarrow$
$\Longrightarrow$
$\Longrightarrow$

As a reminder, here are our one-step evaluation rules for `let` and lambda:

- `let v = e1 in e2` $\Longrightarrow$ `[e1/v]e2`
- `(\v -> e2) e1` $\Longrightarrow$ `[e1/v]e2`

In both cases, `[e1/v]e2` means, "`e1` replaces `v` in `e2`".

3 Practice with recursion

Here is the Nat type from Tuesday's lecture. It says that a `Nat` is either `Z` (zero), or the `S` (successor) of a `Nat`.

```
data Nat = Z | S Nat
  deriving Show
```

And here's the `toInt` function we defined then. It takes a `Nat` to an `Int` by defining a base case and a recursive step. The recursive step for `n` assumes that we can compute `toInt (subOne n)`. Once we have this number, we know that `toInt n` has to be `1` greater than it.

```
toInt :: Nat -> Int
toInt Z = 0
toInt n = 1 + (toInt (subOne n))

subOne (S n) = n -- referenced in toInt, from the lecture slides
```

(3) a. Define a `toNat` function that goes in the opposite direction, converting an `Int` to a `Nat`. I've started you off by giving the base case. Hint: this function will be **very** similar to `toInt` (technical jargon: they are *inverses*). You can check your answer by refreshing `ghci` with `:r`, and then calling, say, `toNat 5`. (Watch your parentheses: in general, `a b c` is grouped as `(a b) c`, not `a (b c)`!)

```
toNat :: Int -> Nat
toNat 0 = Z
toNat i = undefined
```

b. Using your definition, walk through the evaluation of `toNat 2`, one step at a time:
`toNat 2`
$\Longrightarrow$
$\Longrightarrow$
$\Longrightarrow$
$\Longrightarrow$
$\Longrightarrow$

(4) a. Define an `add` function that sums two `Nat`s. Hint: for the recursive step, you will find it useful to remember the following fact about addition: `(1 + m) + n == m + (1 + n)`. You can check your answer by refreshing `ghci` with `:r`, and then calling, say, `add (S (S (S Z))) (S (S Z))`.

```
add :: Nat -> Nat -> Nat
add Z n     = undefined
add (S m) n = undefined
```

b. Using your definition, walk through the evaluation of `add (S (S Z)) (S Z)`.

`add (S (S Z)) (S Z)`
$\Longrightarrow$
$\Longrightarrow$
$\Longrightarrow$

4 Recursion on lists

Recall the type we rolled for `IntList`s: an `IntList` is either `Empty`, or the `Cons` of an `Int` onto an `IntList`:

```
data IntList = Empty | Cons Int IntList
  deriving Show
```

(5) Define `concatIntList`, such that concatenating `Cons 1 (Cons 2 Empty)` and `Cons 3 (Cons 4 Empty)` gives `Cons 1 (Cons 2 (Cons 3 (Cons 4 Empty)))`. Big hint: this definition will be **extremely** close to the definition of `add` you gave above (at least if you followed the hint I gave there!). The reason: for the recursive case, if we know how to compute `concatIntList xs ys`, then concatenating the slightly longer list `Cons x xs` with `ys` is just Consing/adding `x` to `concatIntList xs ys`.

```
concatIntList :: IntList -> IntList -> IntList
concatIntList Empty ys       = ys
concatIntList (Cons x xs) ys = undefined
```

Now convert your definition into one that works on Haskell's native representation of lists, recalling that `Empty` is the `IntList` version of `[]`, and `Cons x xs` is the `IntList` version of `x:xs`. You are re-implementing Haskell's `(++)`.

```
concatList :: [Int] -> [Int] -> [Int]
concatList []     ys = ys
concatList (x:xs) ys = undefined
```

(6) Finally, define a `plus10` function that takes a list as an input and returns a list that's like the old one, but with every number increased by `10`. Test your definition by entering `plus10 [1..10]` into `ghci`.

```
plus10 :: [Int] -> [Int]
plus10 []     = []
plus10 (x:xs) = undefined
```