# LING185A, Assignment #2

Due date: Wed. 1/24/2018

Download the file `Assignment02_Stub.hs` from the CCLE site, and rename it to `Assignment02.hs` (please be careful to use this name exactly). You will need to submit a modified version of this file on CCLE, *and also* a written answer to the last question (either online as a PDF file, or in class as a hard copy).

## Background

A couple of quick notes on Haskell syntax.

- Recall that the predefined type `Bool` has two values, `True` and `False`. The way this type works if exactly analogous to the way the `Shape` type works, except it only has two values rather than three. So it should be obvious that we can construct `case` statements to "split" or "branch" on a `Bool`, which will work according to the following evaluation rules:

$$\texttt{case True of \{True -> } e_1\texttt{; False -> } e_2\texttt{\}} \Longrightarrow e_1$$
$$\texttt{case False of \{True -> } e_1\texttt{; False -> } e_2\texttt{\}} \Longrightarrow e_2$$

  Haskell provides a handy shorthand for this: instead of `case` $e$ `of {True ->` $e_1$`; False ->` $e_2$`}`, you can (if you wish) just write `if` $e$ `then` $e_1$ `else` $e_2$. Under the hood though, it's just another `case` statement.

- When you're giving a name to a lambda-expression (whether in a file or in the first part of a `let` statement), you can write `f x =` $e$ instead of `f = \x ->` $e$. So for example, the following two ways of defining the `isZero` function are completely equivalent:

```
isZero = \n -> case n of {Z -> True; S n' -> False}

isZero n = case n of {Z -> True; S n' -> False}
```

Notice that neither of these points changes *what we can do* using the language — they just give us slightly more convenient ways to write some of the things that we could already write.

## 1 Recursive functions on the `Numb` type

The stub file that you downloaded contains the definition of the `Numb` type for natural numbers that we discussed in class. I've defined some names `one`, `two`, `three`, etc. to be the `Numb`-type representations of these numbers, which is convenient for testing. The `add` function is also repeated here.

**A.** Write a function `sumUpTo :: Numb -> Numb` which computes the sum of all the numbers less than or equal to the given number. For example, given (our representation of) 4, the result should be (our representation of) 10, since $0+1+2+3+4=10$. Here's what you should be able to see in ghci once it's working.

```
*Assignment02> sumUpTo four
S (S (S (S (S (S (S (S (S (S Z)))))))))
*Assignment02> sumUpTo two
S (S (S Z))
*Assignment02> sumUpTo Z
Z
```

**B.** Write a function `equal :: Numb -> (Numb -> Bool)` which returns `True` if the two numbers given are equal, and `False` otherwise.

```
*Assignment02> equal two three
False
*Assignment02> equal three three
True
*Assignment02> equal (sumUpTo three) six
True
*Assignment02> equal (sumUpTo four) (add five five)
True
```

**C.** Write a function `difference :: Numb -> (Numb -> Numb)` which computes the absolute value of the difference between the two given numbers.

```
*Assignment02> difference six four
S (S Z)
*Assignment02> difference four six
S (S Z)
*Assignment02> difference six six
Z
*Assignment02> difference Z three
S (S (S Z))
```

## 2 Recursive functions on lists of `Numbs`

The stub file also defines a `NumbList` type, which represents lists of numbers. Just as a `Numb` is either zero or "one bead on top of" some other `Numb`, a `NumbList` is either the empty list or "a number on top of" some other `NumbList`; we say that the number at the "top", in this sense, is the first element of the list, and the number at the "bottom" (i.e. the innermost number) is the last element of the list. (It's exactly analogous to the `ShapeList` type we used in class; you can see code for that in `Recursion.hs` on the CCLE site.) Some sample lists `list0`, `list1` and `list2` are also defined.

**D.** Write a function `total :: NumbList -> Numb` which computes the total sum of the elements of the given list.

```
*Assignment02> total list0
S (S (S (S (S (S Z)))))
*Assignment02> total list1
S (S (S (S (S (S Z)))))
*Assignment02> total list2
S (S (S (S (S (S (S (S (S (S (S (S (S (S Z)))))))))))))
```

**E.** Write a function `incrementAll :: Numb -> (NumbList -> NumbList)` which adds the given number to each of the elements in the given list.[1]

```
*Assignment02> incrementAll one list0
NonEmptyNL (S (S Z)) (
    NonEmptyNL (S (S (S Z))) (
        NonEmptyNL (S (S (S (S Z)))) EmptyNL
    )
)
*Assignment02> incrementAll two list0
NonEmptyNL (S (S (S Z))) (
    NonEmptyNL (S (S (S (S Z)))) (
        NonEmptyNL (S (S (S (S (S Z))))) EmptyNL
    )
)
*Assignment02> incrementAll one list1
NonEmptyNL (S (S (S (S (S Z))))) (
    NonEmptyNL (S Z) (
        NonEmptyNL (S (S (S Z))) EmptyNL
    )
)
*Assignment02> incrementAll Z list2
NonEmptyNL (S (S (S (S (S (S Z)))))) (
    NonEmptyNL (S Z) (
        NonEmptyNL (S (S (S Z))) (
            NonEmptyNL (S (S (S (S Z)))) EmptyNL
        )
    )
)
```

**F.** Write a function `addToEnd :: Numb -> (NumbList -> NumbList)` which lengthens the given list by putting the given number at the *end* of the list.

```
*Assignment02> addToEnd four (NonEmptyNL three (NonEmptyNL two EmptyNL))
NonEmptyNL (S (S (S Z))) (
    NonEmptyNL (S (S Z)) (
        NonEmptyNL (S (S (S (S Z)))) EmptyNL
    )
)
*Assignment02> addToEnd four list0
NonEmptyNL (S Z) (
    NonEmptyNL (S (S Z)) (
        NonEmptyNL (S (S (S Z))) (
            NonEmptyNL (S (S (S (S Z)))) EmptyNL
        )
    )
)
```

**G.** Write a function `lastElement :: NumbList -> Numb` which computes the last element of the given list; or, if the list is empty, the result should be `Z`.

```
*Assignment02> lastElement list0
S (S (S Z))
*Assignment02> lastElement list1
```

[1] No, the output won't be nicely formatted like this, I'm just writing things this way so that it's a bit more readable and fits on the page.

```
S (S Z)
*Assignment02> lastElement list2
S (S (S (S Z)))
*Assignment02> lastElement (NonEmptyNL four EmptyNL)
S (S (S (S Z)))
*Assignment02> lastElement (NonEmptyNL four list0)
S (S (S Z))
*Assignment02> lastElement EmptyNL
Z
```

**H.** Write a function `contains :: (Numb -> Bool) -> (NumbList -> Bool)`, such that the result of `contains f list` is `True` if there is at least one element of `list` on which `f` returns `True`, and is `False` otherwise.

```
*Assignment02> contains (\x -> equal four x) list0
False
*Assignment02> contains (equal four) list1
True
*Assignment02> contains (\x -> equal (sumUpTo x) six) list0
True
*Assignment02> contains (\x -> equal (sumUpTo x) six) list1
False
```

**I.** Write a function `remove :: (Numb -> Bool) -> (NumbList -> NumbList)` which removes from the given list all the elements on which the given function returns `True`. (So `remove f l` should return `l` unchanged iff `contains f l` returns `False`.)

```
*Assignment02> remove (equal two) list0
NonEmptyNL (S Z) (NonEmptyNL (S (S (S Z))) EmptyNL)
*Assignment02> remove (\x -> equal (difference x three) one) list0
NonEmptyNL (S Z) (NonEmptyNL (S (S (S Z))) EmptyNL)
*Assignment02> remove (\x -> equal (difference x three) one) list1
NonEmptyNL Z EmptyNL
```

**J.** Write a function `append :: NumbList -> (NumbList -> NumbList)` which produces a new list containing the elements of the two given lists combined in the order given. (Hint: This is a lot like `add`, which makes sense if you think about those beads on the sticks. But be careful of the difference between the two ways of writing `add` that we talked about.)

```
*Assignment02> append list0 list1
NonEmptyNL (S Z) (
    NonEmptyNL (S (S Z)) (
        NonEmptyNL (S (S (S Z))) (
            NonEmptyNL (S (S (S (S Z)))) (
                NonEmptyNL Z (NonEmptyNL (S (S Z)) EmptyNL)
            )
        )
    )
)
*Assignment02> append list1 list0
NonEmptyNL (S (S (S (S Z)))) (
    NonEmptyNL Z (
        NonEmptyNL (S (S Z)) (
            NonEmptyNL (S Z) (
                NonEmptyNL (S (S Z)) (NonEmptyNL (S (S (S Z))) EmptyNL)
```

```
            )
        )
    )
)
```

**K.** Write a function `prefix :: Numb -> (NumbList -> NumbList)`, such that the result of `prefix n l` is the length-`n` prefix of `l`; or, if `n` is greater than the length of `l`, the result should just be `l` itself.

```
*Assignment02> prefix two list0
NonEmptyNL (S Z) (NonEmptyNL (S (S Z)) EmptyNL)
*Assignment02> prefix one list1
NonEmptyNL (S (S (S (S Z)))) EmptyNL
*Assignment02> prefix five list1
NonEmptyNL (S (S (S (S Z)))) (NonEmptyNL Z (NonEmptyNL (S (S Z)) EmptyNL))
*Assignment02> prefix Z list1
EmptyNL
*Assignment02> prefix three list0
NonEmptyNL (S Z) (NonEmptyNL (S (S Z)) (NonEmptyNL (S (S (S Z))) EmptyNL))
```

# 3 Induction and recursion

Have a look at the last section of the class handout on recursion. There's an example proof which establishes, via induction, that the function called `f` there "correctly doubles" its argument, i.e. that for all natural numbers $n$, `f` $(\texttt{S}^n\ \texttt{Z})$ will evaluate to $\texttt{S}^{2n}\ \texttt{Z}$.

Let's suppose now that we have these two functions defined:

```
double = \n -> case n of {Z -> Z; S n' -> S (S (double n'))}

isEven = \n -> case n of
                Z -> True
                S n' -> case n' of {Z -> False; S n'' -> isEven n''}
```

Following the example in the handout as a model, write a proof by induction that, for all natural numbers $n$, `isEven (double` $(\texttt{S}^n\ \texttt{Z})$`)` will evaluate to `True`.