Assignment #2

Due date: Thu. 1/21/2021, 12:00pm

Download `Recursion.hs` and `Assignment02_Stub.hs` from the CCLE site, save them in the same directory, and rename `Assignment02_Stub.hs` to `Assignment02.hs` (please be careful to use this name exactly). The `import` line near the top of this stub file imports all the definitions from `Recursion.hs`; you can use them exactly as you would if they were defined in the same file.

You will submit a modified version of `Assignment02.hs` on CCLE; you should not modify or submit `Recursion.hs`.

1 Recursive functions on the `Numb` type

A. Write a function `mult :: Numb -> (Numb -> Numb)` which computes the product of two numbers. You should use the existing `add` function here, and follow a similar pattern to how we wrote that function. Here's what you should be able to see in ghci once it's working.[1]

```
*Assignment02> mult two three
S (S (S (S (S (S Z)))))
*Assignment02> mult one five
S (S (S (S (S Z))))
*Assignment02> mult two two
S (S (S (S Z)))
*Assignment02> mult two (add one two)
S (S (S (S (S (S Z)))))
```

B. 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 $0 + 1 + 2 + 3 + 4 = 10$.

```
*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
```

C. Write a function `equal :: Numb -> (Numb -> Bool)` which returns `True` if the two numbers given are equal, and `False` otherwise. (Hint: For this one you need to work recursively on two `Numb`s, like the way `lessThanOrEq` does.)

```
*Assignment02> equal two three
False
```

[1] Once you've done `mult` and considered its relationship to `add`, you might have the feeling that an exponentiation function is screaming out at you to be written. And once you've written that, you might feel like there's another one screaming out to be written. To understand what's screaming at you, see `https://en.wikipedia.org/wiki/Knuth%27s_up-arrow_notation`.

```
*Assignment02> equal three three
True
*Assignment02> equal (sumUpTo three) (S five)
True
*Assignment02> equal (sumUpTo four) (add five five)
True
```

D. Write a function `bigger :: Numb -> (Numb -> Numb)` which returns the bigger of the two numbers given.

```
*Assignment02> bigger five three
S (S (S (S (S Z))))
*Assignment02> bigger three five
S (S (S (S (S Z))))
*Assignment02> bigger two Z
S (S Z)
*Assignment02> bigger one one
S Z
```

2 Recursive functions on lists

E. Write a function `count :: (Int -> Bool) -> ([Int] -> Numb)` which returns (in the form of a `Numb`) the number of elements in the given list-of-integers for which the given argument returns `True`. (Notice that this is a bit like the `contains` function.)

```
*Assignment02> count (\x -> x > 3) [2,5,8,11,14]
S (S (S (S Z)))
*Assignment02> count (\x -> x < 10) [2,5,8,11,14]
S (S (S Z))
*Assignment02> count (\x -> x < 10) [2,12,3,4,13,14,5,6]
S (S (S (S (S Z))))
```

F. Write a function `listOf :: Numb -> (Shape -> [Shape])` which returns a list containing the given element the given number of times (and nothing else).

```
*Assignment02> listOf four Rock
[Rock,Rock,Rock,Rock]
*Assignment02> listOf three Scissors
[Scissors,Scissors,Scissors]
*Assignment02> listOf (add three two) Paper
[Paper,Paper,Paper,Paper,Paper]
*Assignment02> listOf Z Paper
[]
```

G. Write a function `addToEnd :: Shape -> ([Shape] -> [Shape])` such that `addToEnd x l` returns a list which is like `l` but has an additional occurrence of `x` at the end.

```
*Assignment02> addToEnd Scissors [Rock,Paper]
[Rock,Paper,Scissors]
*Assignment02> addToEnd Rock [Paper,Paper,Paper,Paper]
[Paper,Paper,Paper,Paper,Rock]
*Assignment02> addToEnd Paper []
[Paper]
```

H. Write a function `remove :: (Int -> Bool) -> ([Int] -> [Int])` such that `remove f l` returns a list which is like `l` but with those elements for which `f` returns `True` removed. (Hint: A common mistake here is to think about the task as *changing* the input list into a new list. But that's not what needs to happen at all.[2] The task is to construct a *new list* in a way that depends on, or is "guided by", the contents of the input list.)

```
*Assignment02> remove (\x -> x > 3) [2,5,8,11,14]
[2]
*Assignment02> remove (\x -> x < 10) [2,5,8,11,14]
[11,14]
*Assignment02> remove (\x -> x < 10) [2,12,3,4,13,14,5,6]
[12,13,14]
```

I. Write a function `prefix :: Numb -> ([Shape] -> [Shape])`, such that `prefix n list` returns the list containing the first `n` elements of `list`; or, if `n` is greater than the length of `list`, returns `list` as it is. (Hint: For this one you need to work recursively on two arguments, like the way `lessThanOrEq` works recursively on two `Numb` arguments.)

```
*Assignment02> prefix one [Rock,Paper,Scissors]
[Rock]
*Assignment02> prefix two [Rock,Paper,Scissors]
[Rock,Paper]
*Assignment02> prefix three [Rock,Paper,Scissors]
[Rock,Paper,Scissors]
*Assignment02> prefix four [Rock,Paper,Scissors]
[Rock,Paper,Scissors]
*Assignment02> prefix Z [Rock,Paper,Scissors]
[]
```

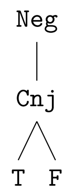
3 Recursive functions on the `Form` type

J. Write a function `countNegs :: Form -> Numb` which returns the number of occurrences of negation in the given formula. The `add` function is helpful here.

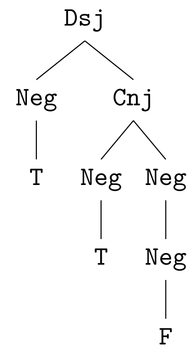
```
*Assignment02> countNegs (Neg (Cnj T F))
S Z
*Assignment02> countNegs (Neg (Neg (Neg (Neg F))))
S (S (S (S Z)))
*Assignment02> countNegs (Cnj (Neg T) (Neg (Neg F)))
S (S (S Z))
*Assignment02> countNegs (Dsj (Neg T) (Cnj (Neg T) (Neg (Neg F))))
S (S (S (S Z)))
```

K. We can represent the structure of a `Form` with a tree, as illustrated by the following examples:

[2] In fact, the whole idea of changing a list into another list is really nonsense, in this setting. Trying to "change `[2,5,8,11,14]` into `[11,14]`" would be just as silly as trying to "change `3` into `4`".



Neg (Cnj T F)



Dsj (Neg T) (Cnj (Neg T) (Neg (Neg F)))

Write a function `depth :: Form -> Numb` which returns the length of the longest root-to-leaf sequence of nodes in the tree for the given formula, i.e. the depth of the most deeply-embedded leaf of the tree. In a one-node tree, this is one. The `bigger` function is useful here.

```
*Assignment02> depth (Neg (Cnj T F))
S (S (S Z))
*Assignment02> depth (Neg (Neg (Neg (Neg F))))
S (S (S (S (S Z))))
*Assignment02> depth (Cnj (Neg T) (Neg (Neg F)))
S (S (S (S Z)))
*Assignment02> depth (Dsj (Neg T) (Cnj (Neg T) (Neg (Neg F))))
S (S (S (S (S Z))))
```

L. Write a function `leftmostLeaf :: Form -> Form` which returns the leftmost leaf node of the tree for the given formula. Notice that this should only ever be either `T` or `F`.

```
*Assignment02> leftmostLeaf (Neg (Cnj T F))
T
*Assignment02> leftmostLeaf (Neg (Neg (Neg (Neg F))))
F
*Assignment02> leftmostLeaf (Cnj (Neg T) (Neg (Neg F)))
T
*Assignment02> leftmostLeaf (Dsj (Neg T) (Cnj (Neg T) (Neg (Neg F))))
T
*Assignment02> leftmostLeaf F
F
*Assignment02> leftmostLeaf (Neg F)
F
```