Assignment #2

Due date: Fri. 1/24/2020, 5:00pm

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.

For the first three sections of this assignment, you will submit a modified version of `Assignment02.hs` on CCLE; you should not modify or submit `Recursion.hs`. For the last section, submit brief written answers either (i) as a PDF or plain text file on CCLE (preferred), or (ii) as a hard copy in class.

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.

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

2 Recursive functions on lists

Your definitions of these functions must *not* use any of Haskell's predefined list functions (`map`, `length`, `filter`, etc., or any of their synonyms); you should use recursion on the structure of the list, like the `total2` and `contains` functions that we saw in class.

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

```
*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 isOdd [two, three, four]
S Z
*Assignment02> count isOdd [two, three, five]
S (S Z)
*Assignment02> count (\x -> x) [True, False, True, True]
S (S (S Z))
*Assignment02> count denotation [f1, Neg f1]
S Z
*Assignment02> count denotation [f1, Neg f1, Dsj f1 (Neg f1)]
S (S Z)
```

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

```
*Assignment02> listOf four True
[True,True,True,True]
*Assignment02> listOf three 10
[10,10,10]
*Assignment02> listOf (add three two) (add one one)
[S (S Z),S (S Z),S (S Z),S (S Z),S (S Z)]
*Assignment02> listOf five []
[[],[],[],[],[]]
```

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

```
*Assignment02> addToEnd 2 [5,5,5,5]
[5,5,5,5,2]
*Assignment02> addToEnd 3 []
```

```
[3]
*Assignment02> addToEnd True [False, False, True]
[False,False,True,True]
```

G. Write a function `remove :: (a -> Bool) -> [a] -> [a]` 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 isOdd [two, three, four]
[S (S Z),S (S (S (S Z)))]
*Assignment02> remove isOdd [two, three, five]
[S (S Z)]
*Assignment02> remove (\x -> x) [True, False, True, True]
[False]
```

H. Write a function `prefix :: Numb -> [a] -> [a]`, 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 both arguments, like the way `difference` works recursively on both of its `Numb` arguments.)

```
*Assignment02> prefix one [2,5,8]
[2]
*Assignment02> prefix two [2,5,8]
[2,5]
*Assignment02> prefix three [2,5,8]
[2,5,8]
*Assignment02> prefix four [2,5,8]
[2,5,8]
*Assignment02> prefix Z [2,5,8]
[]
```

3 Recursive functions on the `RegExp` type

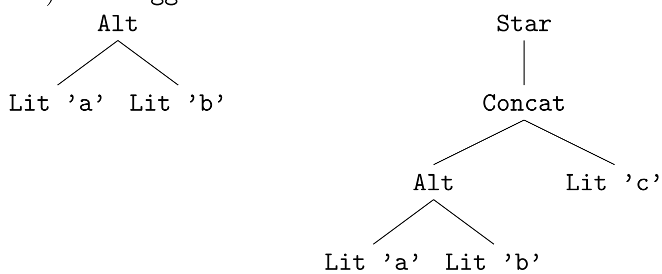
I. Write a function `countStars :: RegExp -> Numb` which returns the number of occurrences of the star operator in the given regular expression. The `add` function is helpful here. (The regular expressions `re17a` and `re17c` are defined in `Recursion.hs`.)

```
*Assignment02> re17a
Alt (Lit 'a') (Lit 'b')
*Assignment02> re17c
Star (Concat (Alt (Lit 'a') (Lit 'b')) (Lit 'c'))
*Assignment02> countStars re17a
Z
*Assignment02> countStars re17c
```

[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`".

```
S Z
*Assignment02> countStars (Concat (Star re17a) (Star re17c))
S (S (S Z))
*Assignment02> countStars (Star (Star (Star (Star (Star (Lit 'a'))))))
S (S (S (S (S Z))))
```

J. We can represent the structure of a `RegExp` with a tree, as shown below for `re17a` and `re17c`. Write a function `depth :: RegExp -> Numb` which returns the length of the longest root-to-leaf sequence of nodes in this tree for the given regular expression, i.e. the depth of the most deeply-embedded leaf of the tree. In a one-node tree, this is one. (Notice that there is no separate `Lit` node on top of a character; there is exactly one node in the tree for each `RegExp`, and `Lit 'a'` is a `RegExp` but `'a'` is not.) The `bigger` function is useful here.




```
*Assignment02> depth re17a
S (S Z)
*Assignment02> depth re17c
S (S (S (S Z)))
*Assignment02> depth (Concat (Star re17a) (Star re17c))
S (S (S (S (S (S Z)))))
*Assignment02> depth (Star (Star (Star (Star (Star (Lit 'a'))))))
S (S (S (S (S (S Z)))))
*Assignment02> depth ZeroRE
S Z
```

K. Write a function `reToString :: RegExp -> [Char]` which returns the string — i.e. the list of characters — representing the given regular expression in the notation we used in class, with some obvious simplifications: use a plain asterisk in place of the superscript asterisk, and use a period in place of the "center dot".[3] Don't forget to include all parentheses. You can use the `++` operator here to concatenate strings. (There is no distinction between strings and lists of characters: `"abc"` is just a convenient notation for `['a','b','c']`.)

```
*Assignment02> reToString re17a
"(a|b)"
*Assignment02> reToString re17b
"((a|b).c)"
*Assignment02> reToString re17c
"((a|b).c)*"
*Assignment02> reToString (Concat (Star re17a) (Star re17c))
"((a|b)*.((a|b).c)**)"
*Assignment02> reToString (Star (Star (Star (Star (Star (Lit 'a'))))))
"a*****"
*Assignment02> reToString (Concat ZeroRE OneRE)
"(0.1)"
```

[3] Ideally we would do something more to distinguish `Lit '0'` from `ZeroRE`, and `Lit '1'` from `OneRE`. But we won't bother here. If you're bored sometime, try writing a version of this function that produces appropriate LaTeXcode.

4 Some nice patterns

L. Notice that for any formula ϕ that we pick, its denotation $[\![\phi]\!]$ is going to be equal to $[\![\neg\neg\phi]\!]$. Similarly, $[\![(\phi \wedge \neg\phi)]\!]$ is going to be equal to $[\![\mathbf{T}]\!]$, for any formula ϕ that we choose. When denotations coincide like this we can say that ϕ is equivalent to $\neg\neg\phi$, and that $(\phi \wedge \neg\phi)$ is equivalent to $\mathbf{T}$. For each of the following, say whether it is equivalent to ϕ, equivalent to $\mathbf{T}$, equivalent to $\mathbf{F}$, or not equivalent to any of these.

(a) $(\phi \vee \mathbf{F})$

(b) $(\phi \wedge \mathbf{F})$

(c) $(\phi \wedge \mathbf{T})$

(d) $(\mathbf{T} \vee \mathbf{F})$

(e) $(\mathbf{T} \wedge \mathbf{F})$

M. We can do something analogous for regular expressions. For example, for any regular expression r, its denotation $[\![r]\!]$ is going to be equal to $[\![(r \mid r)]\!]$. (Because $X = X \cup X$, for any set X.) So we can say that r is equivalent to $(r \mid r)$. For each of the following, say whether it is equivalent to r, equivalent to $\mathbf{1}$, equivalent to $\mathbf{0}$, or not equivalent to any of these.

(a) $(r \mid \mathbf{0})$

(b) $(r \cdot \mathbf{0})$

(c) $(r \cdot \mathbf{1})$

(d) $(\mathbf{1} \mid \mathbf{0})$

(e) $(\mathbf{1} \cdot \mathbf{0})$

(Just for fun: (i) Think about $(\phi \wedge (\psi \vee \chi))$. (ii) Can you find some cases where the pattern breaks down?)