Ling 185A: Assignment 2

Dylan Bumford, Winter 2024

due: Wed, Jan 24

1 Recursion on `Nat`s

Here is the `Nat` type from last week. 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. It turns a `Nat` into an `Int` by defining a base case and a recursive step. The recursive step for `S n` assumes that we can compute `toInt n`. Since `S n` is the successor of `n`, we know `toInt (S n)` should be `1` greater than `toInt n`.

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

(1) Define a function `toNat` that goes in the opposite direction, converting an `Int` to a `Nat`. You can check your answer by loading (or refreshing) your module in ghci, and then calling, say, `toNat 5`.

```
toNat :: Int -> Nat
toNat 0 = undefined
toNat n = undefined
```

(2) Here we're going to define some arithmetic functions on `Nat`s. It would completely defeat the purpose of the exercise if you were to use `toInt` and/or `toNat` in your definitions, so please do not. But you can use them when testing your functions to see if they do what you expect.

a. Define a function `add` 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))`, or more transparently `toInt (add (toNat 3) (toNat 2))`.

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

b. Define a function `mul` to multiply two `Nat`s. You will likely find it useful to make use of the `add` function above.

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

c. Define a function `equal` that determines whether two `Nat`s are equal.

```
equal :: Nat -> Nat -> Bool
equal = undefined
```

2 Recursion on lists

Here is the type we defined 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
```

(3) Define `concatIntList` that concatenates two `IntList`s. That is, the result of `concatIntList u v` should be a single `IntList` containing first all the elements of `u` and then all the elements of `v`. For instance, applying `concatIntList` to `Cons 1 (Cons 2 Empty)` and `Cons 3 (Cons 4 Empty)` should give `Cons 1 (Cons 2 (Cons 3 (Cons 4 Empty)))`. You may feel that this is eerily similar to the definition of `add` you wrote above.

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

Now convert your definition into one that works on Haskell's native representation of lists, recalling that for Haskell's lists, `Empty` is pronounced `[]`, and `Cons x xs` as `(x : xs)`

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

(4) Define a function `count` to count how many of the elements in a list of `Int`s satisfy some property `p`. Since a count is always a natural number, have your function return a `Nat`, as in the type specified below.

```
count :: (Int -> Bool) -> [Int] -> Nat
count p []     = undefined
count p (x:xs) = undefined
```

For instance, you should get these results in ghci.

```
ghci> count (\x -> x > 3) [2, 5, 8, 11, 14]
S (S (S (S Z)))
ghci> count (\x -> x < 10) [2, 5, 8, 11, 14]
S (S (S Z))
```

(5) Define a function `append` that when given a character `c` and a string `u` returns a string just like `u` but with `c` tacked on to the end.

```
append :: Char -> String -> String
append c ""     = undefined
append c (u:us) = undefined
```

For instance, you should get these results in ghci.

```
ghci> append 'x' "this"
"thisx"
ghci> append 'y' ""
"y"
```

(6) Define a function `reverse` that when given a string `u` returns a string just like `u` but with all the characters in the opposite order. You may want to use the `append` function you just defined here.

```haskell
reverse :: String -> String
reverse ""     = undefined
reverse (u:us) = undefined
```

For instance, you should get these results in `ghci`.

```
ghci> reverse "tomorrow"
"worromot"
ghci> reverse "omorrow"
"worromo"
```

3 Regular Expressions

In this section, you will make use of the `Regexp.hs` module that has been imported above. When building `Regexp`s, I recommend using the functions `str`, `char`, `(<.>)`, `(<|>)`, and `rep`, rather than the data constructors `Lit`, `Cat`, `Alt`, and `Star` directly, as these functions will be a little faster and more readable. Feel free to check your work in `ghci` by using the `match` and/or `mset` functions from the library.

(7) Define a function `anyOf` that turns a list of characters into a `Regexp` that matches any string consisting of exactly one character from the list. For instance, `anyOf ['a','b','c']` should match `"a"`, `"b"`, and `"c"`, but nothing else. In other words `mset (anyOf ['a','b','c'])` should return `["a","b","c"]`.

```haskell
anyOf :: [Char] -> Regexp
anyOf []     = undefined
anyOf (c:cs) = undefined
```

(8) Once you've defined `anyOf`, these you should be able to use the following `Regexp`s to pick out various character classes:

```haskell
anych, alpha, lower, upper, digit :: Regexp
anych = anyOf (['!'..'~'] ++ " \n\r\t") -- matches any single character
lower = anyOf ['a'..'z'] -- matches any lowercase letter
upper = anyOf ['A'..'Z'] -- matches any uppercase letter
alpha = lower <|> upper -- matches any letter
digit = anyOf ['0'..'9'] -- matches any digit
```

For instance, `alpha` matches any string consisting of a single alphabetical letter. Using these definitions, together with the other functions from `Regexp.hs`, define `Regexp`s for the following string patterns.

a. Any alphanumeric string that begins with uppercase S.

```haskell
startsWithS :: Regexp
startsWithS = undefined
```

b. Any alphanumeric string with an even number of letters (0, 2, 4, …).

```haskell
evenLetters :: Regexp
evenLetters = undefined
```

c. Any string that begins with an uppercase letter followed by any number of lowercase letters.

```haskell
capitalized :: Regexp
capitalized = undefined
```

d. Any alphabetical string that begins or ends with a lowercase z.

```haskell
termZ :: Regexp
termZ = undefined
```

e. Any string with at least one `i` or `I`.

```haskell
oneI :: Regexp
oneI = undefined
```

(9) Write a function `showRE` that displays a `Regexp` in a string format approximating its mathematical notation. Instead of the emptyset symbol for `Zero`, you can use the representation "0" and instead of the epsilon symbol for `One`, just use the representation "1". For alternation, use a pipe "|", for concatenation a period ".", and for repetition, an asterisk "*". Pay attention to parentheses. This remind you of the `showForm` function from the slides.

```haskell
showRE :: Regexp -> String
showRE re = case re of
  Zero     -> undefined
  One      -> undefined
  Lit c    -> undefined
  Alt r s -> undefined
  Cat r s -> undefined
  Star r  -> undefined
```

```
ghci> showRE (Alt (Star (Lit 'a')) (Cat (Star (Alt (Lit 'b') (Lit 'c'))) One))
"(a* | ((b | c)* . 1))"

ghci> showRE (Cat (Cat Zero (Star (Alt (Lit 'b') ( Lit 'c')))) (Lit 'd'))
"((0 . (b | c)*) . d)"
```

(Note that when creating test examples for this problem, you should use the actual `Regexp` data constructors, as I've done here. The `(<.>)`, `(<|>)`, and `rep` operators exploit the algebraic properties of REs to simplify them, but here you want to actually see them in their raw, unsimplified form. I mean, if you're curious, you should try some examples with both versions and see the difference!)