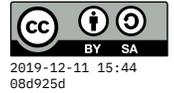


Notes from 11 Sep



Recursive numeric

The notes from 9/9 have a recursive implementation of pow that loops (and multiplies) n times if the exponent is n . (You can see the trace of pow 2 6 in those notes.)

We can do much better using a “fast exponentiation” algorithm, which distinguishes between even and odd exponents. Here it is:

```
square x = x*x
```

```
fpow x n
  | n < 0 = error "Negative argument"
  | n == 0 = 1
  | even n = fpow (square x) (n `div` 2)
  | otherwise = x * fpow x (n-1)
```

Here is a trace of this faster technique, on the same arguments:

```
fpow 2 6 [] [even]
fpow (square 2) 3 []
fpow 4 3 [] [odd]
4 * fpow 4 2 [] [even]
4 * fpow (square 4) 1 []
4 * fpow 16 1 []
4 * 16 * fpow 16 0 [] [zero]
4 * 16 * 1 []
64 * 1 []
64
```

More flexible palindrome detection

One simple solution to the palindrome problem on assignment 1 is just this:

```
palindrome str =
  str == reverse str
```

That works fine for racecar, but it might be nice to detect palindromes where the case, spacing, and punctuation vary. Here’s what we came up with:

```
palindrome str =
  s == reverse s
  where
    s = filter isAlpha (map toLower str)
```

Usage:

```

[]> palindrome "Was it a car or a cat I saw?"
True

```

More sophisticated ciphers

We pursued a generalization of `rot13` on the assignment, where the number of steps to rotate could be a parameter.

```
alpha = ['a'..'z']
```

```

rot k c =
  if little `elem` front
  then chr (ord c + safeK)
  else if little `elem` back
       then chr (ord c - diff)
       else c
  where
    little = toLower c
    safeK = k `mod` 26
    diff = 26 - safeK
    (front, back) = splitAt diff alpha

```

The `splitAt` is just doing take and drop at the same time. The `safeK` represents the offset `k` being clamped to the range `[0..25]`. Let's look at the example where `k` is 5 (thus, `safeK` is also 5 and `diff` is 21).

```

[]> splitAt 21 alpha
("abcdefghijklmnopqrstu", "vwxyz")
[]> chr (ord 'e' + 5)
'j'
[]> chr (ord 's' + 5)
'x'
[]> chr (ord 'x' - 21)
'c'

```

It's always moving forward by 5 characters. If we start in the back segment of the alphabet (as with `'x'`), then the effect of subtracting 21 is rotating around.

We can reproduce the effect of the hard-coded `rot 13` with `(rot 13)`:

```

[]> map (rot 13) "Hello"
"Uryyb"
[]> map (rot 13) "Uryyb"
"Hello"

```

But we can also use other pairs to encode/decode, including negative numbers.

```

[]> map (rot 5) "Testing"

```

```
"Yjxyntl"  
□> map (rot (-5)) "Yjxyntl"  
"Testing"
```

The next step we took still deserves some explanation, but I'll reproduce it here. We don't **need** to keep the rotation offset constant, but instead it can change with each character:

```
□> zipWith rot [1..] "apple"  
"brspj"
```

This rotates the first letter by one position, second letter by two, third letter by three, etc. That way, the two 'p' characters in "apple" actually encode to different letters (rs).

A more secure code would use a **random** sequence of numbers that either *never repeats* (called a one-time pad) or that repeats with a long period. Let's say you and your friend randomly choose this sequence of numbers:

```
□> secret = cycle [21, 13, 2, 5, 11, 6, 25, 8]
```

The cycle function just repeats those over and over:

```
□> take 20 secret  
[21,13,2,5,11,6,25,8,21,13,2,5,11,6,25,8,21,13,2,5]
```

Now you can encode and decode a secret message like this:

```
□> zipWith rot secret "Cryptography is really fun"  
"Xeaueufzvcjd or mrcqwe npa"  
□> zipWith rot (map negate secret) "Xeaueufzvcjd or mrcqwe npa"  
"Cryptography is really fun"
```