1 Infinite lists and lazy evaluation

4.1 Consider two series (i.e. infinite sums) that converge to π:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots$$
$$\pi = 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \cdots$$

Write two functions `calcPi1, calcPi2 :: Int -> Double` that compute π approximately using a given number of terms; investigate which of the series converges faster.

Hint: Express the series as infinite lists for the numerators and denominators separately and combine them using `zip` or `zipWith`.

4.2 *Twin primes* are pairs of consecutive prime numbers (p, p') such that $p' = p + 2$. For example: $(3, 5)$ and $(5, 7)$ are twin primes (because $5 = 3 + 2$ and $7 = 5 + 2$) but $(7, 11)$ are not twin primes (because $11 \neq 7 + 2$).

The *twin prime conjecture* states that there exists an infinity number of twin primes, but this has not been yet been proved.[1].

We can, however, test this for "small" numbers experimentally using a Haskell program. Define the (infinite?) list `twinPrimes :: [(Integer,Integer)]` of all twin primes pairs by ascending order.

Hint: Use the definition of the infinite list of primes presented in the lectures.

4.3 The *Hamming numbers* are generated by starting at 1 and multiplying by 2, 3 or 5. Another way of characterizing them is saying that Hamming numbers have the form $2^i \times 3^j \times 5^k$ for i, j, k non-negative integers.

We can use a Haskell list comprehension to generate some Hamming numbers:

```
ghci> [2^i*3^j*5^k | i<-[0..2], j<-[0..2], k<-[0..2]]
[1,5,25,3,15,75,9,45,225,2,10,50,6,30,150,18,90,450,4,20,100,
12,60,300,36,180,900]
```

However, the following expression does *not* produce the infinite list of all Hamming numbers (why?):

```
[2^i*3^j*5^k | i<-[0..], j<-[0..], k<-[0..]]
```

Write a correct expression to generate the infinite list of Hamming numbers. *Hint*: start by an auxiliary definition to generate numbers of the form $2^i \times 3^j \times 5^k$ such that $i + j + k = n$ for some given n.

4.4 This exercise asks you to implement a method for producing Hamming numbers in ascending order and without repetitions.

[1] `https://en.wikipedia.org/wiki/Twin_prime`

(a) Define a function `merge :: [Integer] -> [Integer] -> [Integer]` for merging two infinite ordered lists, maintaing the order and removing duplicates.

(b) Define the infinite list `hamming :: [Integer]`: it starts with 1, followed by the merging of the mapping of 2×, 3× and 5× over the `hamming` list recursively.

2 Programming with IO

4.5 *The ROT13 substitution cipher.*

(a) Write a Haskell program to implement the "ROT13" text transformation: exchange every letter with the corresponding one 13 positions ahead; non-letters characters remaing unchanged. This is a very simple substitution cipher that was used in the early internet years to hide "spoilers" in message forums; see `https://en.wikipedia.org/wiki/ROT13`.

Hint: use the `ord` and `chr` functions from the `Data.Char` module to convert charaters to and from integers.

(b) If you followed the "functional core, imperative shell" design, you can test the core of the implementation using a QuickCheck property: applying ROT13 twice to any string should give back the original string.

4.6 We want to format words of a text into a paragraph so that each line does not exceed a maximum given width. We start with a few type synonyms for readablity:

```
type AWord     = String
type Line      = [AWord]
type Paragraph = [Line]
```

A *word* is just a string, a *line* is a list of words and a *paragraph* is a list of lines.[2]

(a) Define a pure function `fillWords :: Int -> [AWord] -> Paragraph` that fills a paragraph given a maximum width and a list of words.
Example:

```
   fillWords 10 ["aa", "b", "ccc", "d", "aa", "bbb"] ==
== [["aa","b","ccc","d"],["aa","bbb"]]
```

(b) Use the previous function to write a complete program that reads words from the standard input and writes out a paragraph with maximum length of (say) 70 characters.[3]

[2] We use `AWord` instead of `Word` to avoid clashing with the Prelude type for unsigned integers.

[3] This limit should be made into a command-line argument, but is hard-coded in this exercise for simplicity.

Hint: some Prelude functions that could be useful for this task: `words`, `unwords`, `lines`, `unlines`. Check the documentation about them.

4.7 *A basic spelling checker.* We want to write a program which reads a dictionary (a list of valid words) and a text and marks words that do not occur in the dictionary (i.e. possible spelling mistakes). We start with a type synonym: a dictionary is simply a list of words (i.e. strings).

```
type Dict = [String]
```

In Linux systems the file `/usr/share/dict/words` contains the "default language" dictionary; this is a large text file with one word in each line. We can read this using the following Haskell code.

```
readDict :: IO Dict
readDict = do txt <- readFile "/usr/share/dict/words"
              return (words txt)
```

(a) As a warm-up exercise, write a main program that calls `readDict` and prints the length of the dictionary (i.e. the number of words). Run this program on your machine to confirm that it works. It should report roughly 100K words.

(b) Write a pure function `checkWord :: Dict -> String -> String` that takes a dictionary and a string and adds "reverse video" escape codes[4] when the string isn't in the dictionary.

```
checkWord ["good","words"] "good" == "good"
checkWord ["good","words"] "bad"  == "\ESC[7mbad\ESC[0m"
```

The sequence `"\ESC[7m"` switches to reverse video and `"\ESC[0m"` switches back to normal video; try `putStrLn (checkWord ...)` in GHCi to see the effect.
Use the Prelude function `elem` to check the occurrence of a string in a list.

(c) Write a pure function `spellCheck :: Dict -> String -> String` that breaks up a text into lines and words, applies the `checkWord` for each word, and joins them back into a complete text again. *Hint*: use `words`, `unwords`, `lines` and `unlines`.

(d) Write the main IO action that reads the dictionary, reads the standard input with `getContents`, and outputs the result of `spellCheck` using `putStr`.

Note: your program will perform sequential search of each word in a large list (the dictionary); we will later see how to implement a more efficient data structure (e.g. a binary search tree) and speed up this program considerably.

[4] https://en.wikipedia.org/wiki/ANSI_escape_code