10 Imperative programming

Exercise 10.1 (Warm-up: word count, `WordCount.hs`). Implement a simplified version of the UNIX command `wc`, see Figure 3. For example, applied to the current snapshot of the practicals, the program displays.

```
ralf@melian ~/Teaching/FP2/practicals $ wc *.hs
  356  2006 13078 FP1.hs
   61   403  2873 Introduction.hs
  291  1295  9235 IO.hs
  419  2537 14580 Lazy.hs
  136  1081  5663 Modules.hs
  117   355  2627 Monads.hs
  118   208  3254 practicals.hs
  127   414  2713 Types.hs
   58    94   978 Unfolds.hs
 1683  8393 55001 total
```

For accessing the program's command line arguments, the standard library `System.Environment` provides the command `getArgs :: IO [String]`. As an example, the program

```
module Main where
import System.Environment
main  ::  IO ()
main  =  do  args ← getArgs
             putStrLn (unwords args)
```

echoes the command line arguments to the standard output. Notice that a stand-alone Haskell program must contain a `Main` module that contains a definition of `main :: IO ()`. To compile the program, use GHC's built-in make facility, `ghc -make`, e.g.

```
ralf@melian $ ghc --make Echo.hs
[1 of 1] Compiling Main             ( Echo.hs, Echo.o )
Linking Echo ...
ralf@melian $ ./Echo hello world
hello world
```

Exercise 10.2 (Mastermind, `Mastermind.hs`). *Mastermind* is a game for two players: the *code-maker* and the *code-breaker*. The goal of the code-breaker is to guess the code word designed by the code-maker. In more detail:

- The *code-maker* thinks of a code word that consists of a sequence of *four* codes, where a single code is chosen from a total of *eight* colours: *white*, *silver*, *green*, *red*, *orange*, *pink*, *yellow*, and *blue*. There are no restrictions on the code word: it may consist of four identical colours, but also of four different colours. Thus, in total there are $8^4 = 4096$ possible combinations to choose from.

- The *code-breaker* tries to guess the code word in as few turns as possible. If they need more than *twelve* guesses, they lose. A turn consists of making a guess, suggesting a code word i.e. a sequence of four colours. The *code-maker* responds by providing two hints:

```
WC(1)                   User Commands                   WC(1)

NAME
       wc - print newline, word, and byte counts for each file

SYNOPSIS
       wc [FILE]...

DESCRIPTION
       Print newline, word, and byte counts for each FILE, and
       a total line if more than one FILE is specified. A word
       is a non-zero-length sequence of  characters  delimited
       by white space.
```

Figure 3: An excerpt of the man page for `wc` (slightly simplified).

   – the number of correct colours in the right positions;
   – the number of colours placed in the wrong position.

Write a console I/O program that implements the game *Mastermind*, where the user takes the role of a *code-breaker* and the computer is the *code-maker*. Try to minimize the I/O part of your program. Furthermore, design the code in such a way that it abstracts away from the various constant values mentioned above: it should work for any code length, for any number of colours, and for any maximum number of tries.

The standard library `System.Random` provides an extensive infrastructure for pseudo-random number generation. As an example use case, the computation `dice` delivers a random integer between `1` and `6`.

```
dice  ::  IO Int
dice  =  getStdRandom (randomR (1,6))

roll  ::  IO Int
roll  =  do  a ← dice
             b ← dice
             return (a + b)
```

Like `dice`, the computation `roll` is not pure: the answer may be different for each invocation:

```
 ⋙ roll
5
 ⋙ roll
12
```

Recall that for expressions of type `IO`, the Haskell interpreter first performs the computation and then prints the resulting value.

Exercise 10.3 (Worked example: one-time pad encryption, `OTP.hs`). The purpose of this exercise is to develop a program that encrypts and decrypts text files using the one-time pad method[9] (OTP for short).

OTP is an encryption method that provides, at least theoretically, perfect security and is thus unbreakable. To encode a message, it is XOR-ed it with a stream of *truly* random numbers. Decryption works in the same way, requiring an exact copy of the original random number stream. In theory, this method is unbreakable. In reality, there are several problems:

- Generating random numbers is far from trivial. Most random number generators found in programming languages are not truly random.
- Every random stream may be used only once, and has to be destroyed afterwards. Destroying data on a public computer is difficult.
- The transmitter and receiver must have an exact copy of the random stream. Therefore, the system is vulnerable to damage, theft, and copying.

Despite these drawbacks, the OTP method has been used, for example, by the KGB, the main security agency of the former Soviet Union. They used truly random streams printed in a very small font, so that it could be easily hidden, see Figure 4.

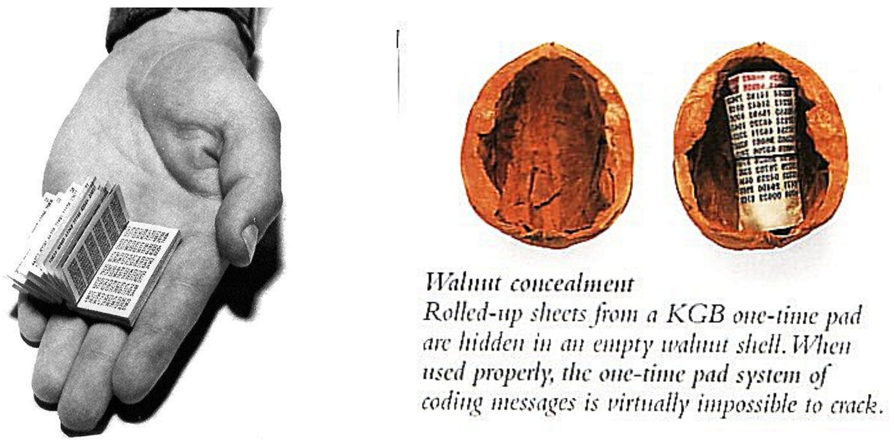

Walnut concealment
Rolled-up sheets from a KGB one-time pad are hidden in an empty walnut shell. When used properly, the one-time pad system of coding messages is virtually impossible to crack.

Figure 4: Truly random stream sources for OTP encryption, used in espionage. Sources: `http://www.ranum.com/security/computer_security/papers/otp-faq/` (left image); `http://home.egge.net/~savory/chiffre9.htm` (right image).

Haskell's global random number generator is initialized automatically in some system-dependent fashion, for example, by using the time of day, or Linux's kernel random number generator. To get *reproducible* behaviour (for en- and decryption) initialize it with some random seed:

```
main = do setStdGen (mkStdGen 4711)
```

Of course, you should keep the seed value secret, top secret. Use the pseudo-random number generator, see Exercise 10.2, to generate a sequence of pseudo random numbers $r_0, r_1, \ldots$.

Encryption and decryption then works as follows. For simplicity and to avoid bit fiddling, we use a Caesar cipher instead of XOR. Let A be the input text file that should be encrypted/decrypted

[9]Source: `http://en.wikipedia.org/wiki/One-time_pad`

to the output text file B. Read the ASCII characters $a_0, a_1, \ldots, a_n$ from A, and write the characters $b_0, b_1, \ldots, b_n$ to B.

$$b_i = \begin{cases} a_i & \textbf{if } a_i < 32 \\ ((a_i - 32 \oplus r_i) \bmod (128 - 32)) + 32 & \textbf{if } a_i \geq 32 \end{cases}$$

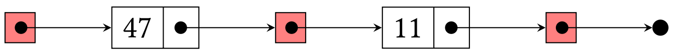
The case distinction ensures that only printable ASCII characters ($32 \leq a_i < 128$) are encrypted, whereas non-printable ones ($0 \leq a_i < 32$) are simply copied. When *encrypting*, $\oplus$ is *addition* (+). When *decrypting*, $\oplus$ is *subtraction* (−).

Write a console I/O program that accepts the input `encrypt` A B, where A and B are paths to text files. It encrypts the contents of A and writes the result to B. The input `decrypt` A B instructs the program to decrypt A, writing the result to B.

Test your program on some important input file A, for example, your bachelor thesis, that you first encrypt and then decrypt. Verify that the decrypted file is identical to A. (If you feel adventurous, delete the input file after encryption and only retain the decrypted variant.)

Exercise 10.4 (Singly-linked lists, `LinkedList.hs`).
Consider the implementation of singly-linked lists shown in the lectures.

```
type ListRef elem  =  IORef (List elem)
data List elem  =  Nil | Cons elem (ListRef elem)
```



1. Define constructor functions:

   ```
   nil   :: IO (ListRef elem)
   cons  :: elem → ListRef elem → IO (ListRef elem)
   ```

   The operation `nil` creates an empty linked list; `cons` attaches an element to the front of a given linked list.

2. Implement conversion functions

   ```
   fromList  :: [elem] → IO (ListRef elem)
   toList    :: ListRef elem → IO [elem]
   ```

   that convert between Haskell's standard lists and singly-linked lists.

3. Define an internal iterator for singly-linked lists:

   ```
   foreach :: ListRef a → (a → IO b) → IO (ListRef b)
   ```

   The call `foreach list action` applies `action` to each element of the singly-linked list `list`, returning a singly-linked list collecting the results. For example,

   ```
    ⋙ as ← fromList [0 .. 3]
    ⋙ bs ← foreach as (\ n → do print n ; return (n + 1))
   0
   1
   2
   ```

```
3
 ⋙ print (toList bs)
[1, 2, 3, 4]
```

Can you modify the implementation so that the elements of the input list are overwritten? Do you have to change `foreach`'s type signature?