10 Imperative programming

Exercise 10.1 (*Warm-up:* word count, `WordCount.hs`). Implement a simplified version of the UNIX command `wc`, see Figure 1. For example, applied to the template files for this week, the program displays:

```
$ wc *.hs
  24  102  588 LinkedList.hs
  41  158 1065 Mastermind.hs
  17   55  365 OTP.hs
  15   50  319 WordCount.hs
  97  365 2337 total
```

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

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

echoes the command line arguments to the standard output.

Also remember `mapM` and `mapM_` (see Hint 1), that can be used to map *IO actions* over lists. To change the above program into one that lists the contents of all the arguments (i.e. do the same thing as the UNIX program `cat`), simply change the `print` action to:

```
  fileContents ← mapM readFile args
  mapM_ putStr fileContents
```

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.

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

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

Exercise 10.2 (*Mandatory:* 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:

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

  For example, if the code word is ‘Red Red Blue Green‘ and the code-breaker guesses ‘Blue Red Yellow Blue‘, then the hint will be that 1 colour is in the correct position, and 1 colour is placed in the wrong position.

In this exercise you will 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, try to 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.

A typical run of the game should look like this, with input from the user in italic:

```
I picked a random code word with 4 colours.
Possible colours are White Silver Green Red Orange Pink Yellow Blue.
Try to guess the secret code word, 12 tries left
> Blue Red Yellow Blue
Incorrect
0 colour(s) in the correct position,
1 colour(s) in the wrong position.
Try to guess the secret code word, 11 tries left
> White Blue Green Pink
Incorrect
2 colour(s) in the correct position,
2 colour(s) in the wrong position.
Try to guess the secret code word, 10 tries left
> Green Blue White Pink
Correct
```

If the code-breaker loses, then the interaction should look like this:

```
Try to guess the secret code word, 1 try left
> Orange Red White Blue
Incorrect
```

```
0 colour(s) in the correct position,
0 colour(s) in the wrong position.
No more tries, game over.
The code was Pink Pink Pink Pink.
```

1. Choose a datatype for representing colours, and one for representing code words

2. Implement a function that, given a secret code and a guess, counts the number of correct colours in the right position and the number of wrongly placed colours. Do you need IO in this function? *(Tip: the number of wrongly placed colours is of course the total number of correct colours regardless of position, minus the number of correct colours in the right position.)*

3. Make a function that generates a new random code word.

   The 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`.

   ```
   roll_d6 :: IO Int
   roll_d6 = randomRIO (1,6)

   roll_2d6 :: IO Int
   roll_2d6 = do
     a ← roll_d6
     b ← roll_d6
     return (a + b)
   ```

   These functions are not pure: the answer may be different for each invocation:

   ```
    ⋙ roll_2d6
   5
    ⋙ roll_2d6
   12
   ```

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

   The random number library doesn't come with minimal installations of GHC. If it is missing, the library can be installed from the command line with

   ```
   cabal install --lib random
   ```

4. Make a function `playGame` that takes a pre-determined code and asks the player for a guess, and displays the number of correctly placed and wrongly placed colours (using your functions from step 2). For some thoughts on parsing user input, see Hint 2.

5. Modify the function `playGame` so it repeats this process for a given number of attempts. It should stop when the code is guessed (and the code-breaker wins) or when there are no more tries (and the code-breaker loses).

6. Add a `main` function that allows the user to play a complete game of *Mastermind* with a randomly generated code.

Exercise 10.3 (*Extra:* Stream ciphers, `OTP.hs`). The purpose of this exercise is to develop a program that encrypts and decrypts text files using the one-time pad method[1] (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 2.

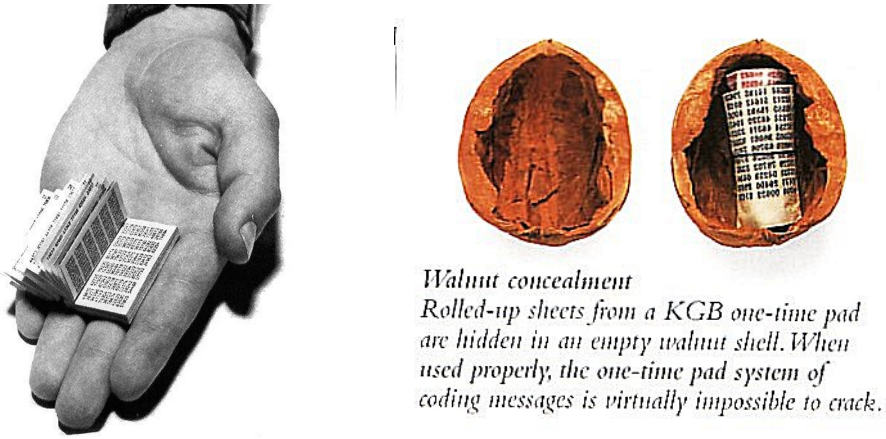

Figure 2: Truly random stream sources for OTP encryption, used in espionage. Sources: http://www.ranum.com/security/computer_security/papers/otp-faq/ (left image); http://home.egge.net/~savory/chiffre9.htm (right image).

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 1337)
```

Of course, this seed value should be kept 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. To avoid bit fiddling, it is possible to use a Caesar-like cipher instead of XOR. Let A be the input text file that should be encrypted/decrypted

[1] 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 appropriate text file *A*. that you first encrypt and then decrypt. Verify that the decrypted file is identical to *A*. If you are looking for something fun to encrypt, http://textfiles.com or Project Gutenberg (https://www.gutenberg.org/) are great resources.

Exercise 10.4 (*Extra:* 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
    ⋙ toList bs
   [1, 2, 3, 4]
   ```

4. Can create a version of `foreach` that actually *overwrites* the elements of the input list? (Note: this will require a slightly different type signature!)

Hints to practitioners 1. Surprise! You can write imperative programs in Haskell with *do-notation*. But is good to remember you never really need *do-notation* (and it also has limitations).

For example, `a >> b` is equivalent to `do { a; b }`, and `a >>= \x → foo x` is equivalent to `do { x ← a; foo x }`. *Do-notation* may more clearly express the 'impure ideas' of a certain part of your program. It is, therefore, very commonly used for the `IO` parts of Haskell programs. At the same time, the *do-notation* shouldn't necessarily *always* be used. For example, `a >>= \x → foo x` can also be written more simply as `a >>= foo`, which neatly reads as 'receive the value from action `a` and send it to the action `foo`'. In this case, the `(>>=)` operator acts like a form of *action composition* comparable to the (hopefully) now-familiar function composition.

As another example, `mapM action [x1, x2, ... xN]` is more or less equivalent to:

```
do y1 ← action x1
   y2 ← action x2
   ...
   yN ← action xN
   return [y1, y2, ... yN]
```

But clearly using `mapM` is less typing and more flexible, and feels quite similar to the purely functional `map` function.

Also note that several IO actions come in two flavours: ones which preserve output, and ones which discard it. The latter ones are typically used when you only care about the *side effect* of an action, and have identifiers ending in an underscore, e.g.

```
sequence  :: [IO a] → IO [a]
sequence_ :: [IO a] → IO ()

mapM  :: (a → IO b) → [a] → IO [b]
mapM_ :: (a → IO b) → [a] → IO ()
```

Hints to practitioners 2. Dealing with user input is never 'simple'. While it is tempting to use the `Read` type class to parse strings into your own data type, this will give you the most annoying text interface ever invented, for instance:

```
⋙ data Foo = Yes | No | Maybe deriving (Show,Read)
⋙ read "Yes" :: Foo
Yes
⋙ read "Nope" :: Foo
*** Exception: Prelude.read: no parse
⋙ read "yes" :: Foo
*** Exception: Prelude.read: no parse
```

That's right: one small mistake and your game aborts. Imagine this happening when the player has almost guessed the code! Parsing will be a subject of a later week; for now we recommend an ad-hoc solution: for example, only checking if the first letter of a word matches a colour (we are lucky: they are all distinct).

Also remember the `words :: String → [String]` function!