

Functional Programming: Exercise 8

Sheet published: Friday, June 14th

Submission deadline: Wednesday, June 26th, 12:00 noon

Note regarding the public holiday On June 20th, there is a public holiday. We won't have an exercise session on that day. This sheet therefore is for two weeks, the submission deadline is on Wednesday, June 26th, 12:00 noon and it will be discussed the day after.

Exercise 8.1 (Skeleton: `WordCount.hs`). Implement a simplified version of the UNIX command `wc`, see Figure 1. For example, applied to two of the provided files, the program displays:

```
sebastian@laptop ~/Teaching/FP19/ex8 $ wc Echo.hs WordCount.hs
 10  24 154 Echo.hs
 10  25 160 WordCount.hs
 20  49 314 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.

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

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

Exercise 8.2 (Skeleton: `Evaluator.hs`). Recall the expression datatype shown in the lectures. Extend the type and its evaluator by non-deterministic choice.

```

data Expr
  = Lit Integer      — a literal
  | Expr :+: Expr    — addition
  | Expr *: Expr     — multiplication
  | Div Expr Expr    — integer division
  | Expr :?: Expr    — non-deterministic choice

```

The expression $e1{:}? : e2$ either evaluates to $e1$ or to $e2$, non-deterministically. For example,

```

toss :: Expr
toss = Lit 0 :?: Lit 1

```

evaluates either to 0 or to 1. To illustrate, here are some example evaluations:

```

>>> evalN toss
[0,1]
>>> evalN (toss :+: Lit 2 *: toss)
[0,2,1,3]
>>> evalN (toss :+: Lit 2 *: (toss :+: Lit 2 *: (toss :+: Lit 2 *: toss)))
[0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]

```

As you can see, the evaluator returns a list of all possible results. Implement this function

```
evalN :: Expr -> [Integer]
```

Haskell's list datatype is already an instance of *Functor*, *Applicative*, and *Monad*. So, all you have to do is to extend the interpreter in applicative style by adding one equation for $:?$: (you can copy the remaining equations from `evalA`).

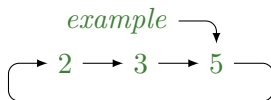
Exercise 8.3 (Skeleton: `Necklace.hs`). We introduce the following data type *Necklace* for non-empty cyclic linked lists:

```
type NecklaceRef elem = IORef (Necklace elem)
data Necklace elem = Cons elem (NecklaceRef elem)
```

It is similar to the linked list introduced in the lectures, but there is no case for the empty list. Values of this data type are meant to be cyclic. For example, we can build:

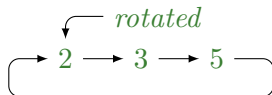
```
example :: IO (NecklaceRef Integer)
example = mdo
  x ← newIORef (Cons 2 y)
  y ← newIORef (Cons 3 z)
  z ← newIORef (Cons 5 x)
  return z
```

We used the *mdo*-syntax¹ since we need mutual recursive assignments. With **do** instead of *mdo*, the first assignment would fail since *y* is not in scope. The result is the following cyclic construct:



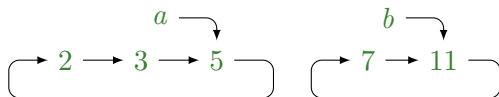
The idea is to represent a list of elements by giving a reference to the *last* element. The example therefore represents the list $[2, 3, 5]$. This representation allows us to easily implement certain operations.

- a) Implement a function *rotate* :: *NecklaceRef* elem → *IO* (*NecklaceRef* elem) that rotates the represented list. To do so, none of the references need to be changed, the result just points to another element in the necklace. The function therefore runs in constant time. With *rotated* = *example* ≫≧ *rotate*, we get:



It represents the list $[3, 5, 2]$. The next rotation represents $[5, 2, 3]$ and the third one again the initial list $[2, 3, 5]$.

- b) Appending two necklaces is possible in constant time, too! Consider *a* and *b* as shown:



To append *a* and *b*, we only need to update two references:



Afterwards, *b* represents $[2, 3, 5] \mathbin{++} [7, 11]$ and *a* represents $[7, 11] \mathbin{++} [2, 3, 5]$. So we do not need to return a result, instead we change the provided arguments. Implement the function *append* :: *NecklaceRef* elem → *NecklaceRef* elem → *IO* ().

¹<https://wiki.haskell.org/MonadFix>

Exercise 8.4. Obtain the free theorems for the following function types:

a) $a \rightarrow (a, a)$

b) $(a, b) \rightarrow (b, a)$

c) $a \rightarrow a$ (can you show that this must be the identity function?)

Exercise 8.5 (Lecture Evaluation - Not graded). Please contribute to the lecture evaluation. Your feedback helps us to improve this course. If you have a `@cs.uni-kl.de` email address, then you already have received an email with access details. If you do not have such an email account, please go to <https://vlu.informatik.uni-kl.de/teilnahme/> and enter your RHRK email address (`username@rhrk.uni-kl.de`). You then will receive an email with access details for the lecture evaluation system.

Access the system using the received URL and then select your lectures. Our lecture is *INF-36-51-V-6: “Funktionale Programmierung”*. Afterwards, you can start answering the questions and give us comments. The evaluation is of course anonymous.

Thanks for taking your time!