- The second set of Haskell exercises comes in two parts: 2A and 2B.
  This is part 2B.
- Use the skeleton `Practicum2B.hs` from the homepage of the course.
- Submit a zip-file containing both `Practicum2A.hs` and `Practicum2B.hs` via `http://cn.dashnet.tech`.
- The goal of the exercises in 2B is to see and prepare a few examples of the use of monads.
- The written exam won't contain questions about monads.
- We made use of earlier notes by Atze van der Ploeg and Joerg Endrullis, also for a course at the VU University.

MaybeOne

Haskell programs are pure mathematical functions and hence have no side effects. Reading and writing, however, can considered as side effects. If functions are allowed to perform input/output, then we can no longer reason about functions as in mathematics. Haskell takes the 'pure' approach, not admitting functions with side effects. Expressions with side effects are called actions and are of type `IO a` which is the type of input/output actions returning a value of type a. Also `IO ()` is admitted, which is the type of actions which do not return any result value. More generally, if a function has side effects (possibly different from input/output) then this must be shown in the type of the function. This is done using *monads*.

We consider first partial functions, that in most cases yield a result, but in some (usually few) cases are undefined, that is, do not yield a result.

An example of such a function is division. It takes as input a number for the numerator n and a number for the denominator d, and returns the n divided by d, that is $\frac{n}{d}$ in case $d \neq 0$. If $d \neq 0$ then the function is undefined.

In order to define this function in Haskell we use the following rendering of a type for the output of partial functions, here called `MaybeOne`. In Haskell it is also predefined as Maybe.

```
data MaybeOne a = NoResult
                | Result a deriving (Show, Eq)
```

Division can now be written as follows:

```
-- Example: partial function for division
myDividedBy :: Double -> Double -> MaybeOne Double
myDividedBy n d =
  if d  == 0
  then NoResult
  else Result (n / d)
```

Here Double is a number-type suitable to use for division. You can try using this function providing it with arguments for which division is defined, and also for example with arguments $n = 1$ and $d = 0$.

Exercise 1

(a) Use `MaybeOne` to define a function

```
myIndexOf :: [Double] -> Double -> MaybeOne Int
myIndexOf l n = undefined
```

that takes as input a list and a potential element of the list, and returns the (first) index where the element occurs (if it occurs). The function is undefined if the potential element does not occur in the list.

Hint: one possibility is to start the definition as follows:

```
myIndexOf l n =
  myIndexOfAux l 1
  where
  myIndexOfAux l i =
```

where the auxiliary function already knows the input `n`, and checks whether it is the head of the list `l` or whether it is possibly in its tail. Of course other solutions are also ok. A call of the function yielding index 3 as result:

```
myIndexOf [4,5,6,7,6,7,6] 6
```

and a call not yielding a result:

```
myIndexOf [1,2,3] 4
```

(b) Use `MaybeOne` to define a function

```
myRemainderString :: String -> String -> MaybeOne String
myRemainderString x y = undefined
```

that takes as input a two strings x and y, and if x is a prefix of y returns what remains of y after removing that prefix, and returns no result otherwise. A possible call of the function yielding a result:

```
myRemainderString "abc" "abcd"
```

and one not yielding a result:

```
 myRemainderString "abc" "accd"
```

Infix functions using case-notation

In Haskell we can write a binary function such as `+,-,*,/` also in infix notation. Such a function is often called an operator. We now make an operator using myDividedBy, as follows:

```
-- Create an operator for our divide function
n // d = n ‘myDividedBy‘ d
```

We can now write `3 // 2`. As a next example we implement the function

$$f(x, y, z) = \frac{\frac{x}{y}}{z}$$

In case $y = 0$, or in case $z = 0$, this function is undefined. In the given Haskell-file, we use next to f also the following function g as an example:

$$g(x, y, z, s) = \frac{\frac{y}{s} + \frac{z}{s}}{\frac{x}{y} - \frac{x}{z}}$$

```
-- Example f using case (1/2)
f :: Double -> Double -> Double -> MaybeOne Double
f x y z = case x // y of
    NoResult           -> NoResult
    Result xDividedByy ->
      case xDividedByy // z of
        NoResult   -> NoResult
        Result   r -> Result r
```

Alternatively we can write this as follows:

```
-- Example f using case (2/2)
fShorter :: Double -> Double -> Double -> MaybeOne Double
fShorter x y z = case x // y of
    NoResult           -> NoResult
    Result xDividedByy -> xDividedByy // z
```

Exercise 2

Consider the following function:

$$v(x, y, z, s) = \frac{\frac{x}{y}}{\frac{z}{s} - \frac{y}{s}} - (\frac{y}{s} + \frac{z}{x})$$

Define this function in Haskell:

```
v1 :: Double -> Double -> Double -> Double -> MaybeOne Double
v1 x y z s = undefined
```

Use the case-syntax as above for f, and as also done in the Haskell-file for the function g. In two exercises below we develop other ways of defining v in Haskell.

Using a new operator

The readability is suboptimal. Note that the following pattern is repeated:

```
 case ... of
  NoResult -> NoResult
  Result ... -> ...
```

This pattern can be written using an operator with notation `>>=`. The operator `>>=` gets a leftResult of type `MaybeOne Double` at the left side, so for example `NoResult` or `Result 10.0`. At the right side it has a function that takes an input of type `Double` and gives back an output of type `MaybeOne Double`. We can use `>>=` to write f better:

```
-- Example f using >==
fBetter :: Double -> Double -> Double -> MaybeOne Double
fBetter x y z = (x // y) >>= dividedByZ
  where dividedByZ xdividedByy = xdividedByy // z
```

or alternatively using lambda-notation as follows:

```
-- Example f using >= and lambda
fBetterLambda :: Double -> Double -> Double -> MaybeOne Double
fBetterLambda x y z = (x // y) >>= (\xDividedByy -> xDividedByy // z)
```

Exercise 3

Consider again the function v from Exercise 2. Give a second rendering `v2` of this function in Haskell, now using the operator `>==`.

```
-- Exercise 3
v2 :: Double -> Double -> Double -> Double -> MaybeOne Double
v2 x y z s = undefined
```

Using do

We can improve the notation as follows:

```
-- Example f using do
fDo :: Double -> Double -> Double -> MaybeOne Double
fDo x y z = do
  xDividedByy <- x // y
  xDividedByy // z
```

Here `xDividedByy <- x // y` means: calculate `x // y` and in case the result is `NoResult` then we stop the computation and return `NoResult` for the complete function. In case the result is `Result num`, then we store the num in the variable `xDividedByy` and continue the calculation. So starting an expression with `do` means that Haskell translates an expression

```
x <- y
nextLine
```

to

```
y >>=
(\x -> nextLine)
```

We can also have an expression with 'do' consisting of several lines. Example:

```
do
x <- y
p <- z
r <- l
lastLine
```

The Haskell compiler translates this into:

```
y >>=
(\x ->
  z >>=
 (\p ->
   l >>=
   (\r -> lastLine)
 )
)
```

Operations having as result a `MaybeOne Double` can be used in combination with operations of another type. Please note: if `let` is used in a block for `do`, then curiously `in` is not written! Example:

```
do
x <- y
let r = q
p <- z
lastLine
```

is translated into:

```
y >>=
(\x ->
  let r = q in
  z >>= (\p -> lastLine)
 )
)
```

We proceed by improving the notation using in addition return:

```
 return :: Double->MaybeOne Double
 return val = Result val
```

Now we can render f in a more readable way, as follows:

```
-- Example f using do-return
fPerfect :: Double -> Double -> Double -> MaybeOne Double
fPerfect x y z = do
  xDividedByy <- x // y
  result      <- xDividedByy // z
  return result
```

Exercise 4

Give a third rendering `v3` of the function v in Haskell, now using the do-notation and return.

```
-- Exercise 4
v3 :: Double -> Double -> Double -> Double -> MaybeOne Double
v3 x y z s = undefined
```