Monada State.
Laborator 10

Reminder

De multe ori avem nevoie să scriem funcții care fac un calcul de natură *imperativă din punct de vedere conceptual*.

Spre exemplu, mai jos avem trei operații asupra stivelor.

```
type Stack = [Int]

push :: Int -> Stack -> ((), Stack)
push x s = ((), x : s)

pop :: Stack -> (Int, Stack)
pop (hd:tl) = (hd, tl)

manip :: Stack -> (Int, Stack)
manip s1 =
  let ((), s2) = push 3 s1 in
    let (x, s3) = pop s2 in
      let (y, s4) = pop s3 in
        (y, s4)
```

Observați că fiecare dintre cele 3 operații actualizează o stivă și întorc în plus și un rezultat. Într-un limbaj pur funcțional, nu se poate actualiza propriu-zis valoarea unei variabile, astfel încât funcțiile de mai sus întorc ca rezultat noua stivă.

Acest lucru face ca secvențierea unor astfel de operații să nu fie deosebit de estetică (vezi funcția `manip`).

În astfel de cazuri, este util să folosim monada `State`, care permite încapsularea calculelor imperative.

> O valoare de tip `State s a` este un calcul imperativ care actualizează o stare de tip `s` și produce la final o valoare de tip `a`.

Putem construi astfel de calcule folosind funcția `state`, așa cum se vede mai jos.

```
import Control.Monad.State.Lazy

pop :: State Stack Int
pop = state $ \(hd:tl) -> (hd, tl)

push :: Int -> State Stack ()
push x = state $ \s -> ((), x : s)
```

Deoarece `State s` este o monadă, sunt definite funcțiile `return` și `>>=`, astfel încât se poate folosi și notația `do`, așa cum se vede mai jos.

```
manip :: State Stack Int
manip = do
  push 3
  x <- pop
  pop
```

În acest moment, calculul are o descriere mult mai naturală.
Pentru a executa un calcul, putem folosi

$$\texttt{runState manip } [6, 7, 4, 9]$$

dacă suntem interesați și de starea finală sau

$$\texttt{evalState manip } [6, 7, 4, 9]$$

dacă ne interesează doar rezultatul.

Implementarea State

Monada `State` poate fi implementată și de utilizator, nefiind strict necesar să folosim implementarea din biblioteca standard.

```
newtype State s a = State { runState :: s -> (a,s) }

instance Functor (State s) where
  fmap f = \(State calcul) -> State (\s -> let (x, s') = calcul s in (f x, s'))

instance Applicative (State s) where
  pure x = State $ \s -> (x, s)
  (State calculf) <*> (State calcula) = State (\s -> let (f, s1) = calculf s in
                                                     let (r, s2) = calcula s1 in
                                                       (f r, s2))


instance Monad (State s) where
  (State h) >>= f = State $ \s -> let (a, newState) = h s
                                      (State g) = f a
                                  in  g newState
type Stack = [Int]

pop :: State Stack Int
pop = State $ \(x:xs) -> (x,xs)

push :: Int -> State Stack ()
push a = State $ \xs -> ((),a:xs)

run :: State s a -> s -> (a, s)
run (State calcul) s = calcul s
```

```
manip :: State Stack Int
manip = do
  push 3
  x <- pop
  pop
```

Exercițiu 1

Implementați algoritmul extins al lui Euclid ca valoare de tip `State s Int`, alegând convenabil o definiție pentru starea `s` (trebuie să țină minte valorile din algoritm care trebuie actualizate).

Exercițiu 2

Scrieți un program care simulează amestecarea unui pachet de 52 cărți de joc. Folosiți algoritmul Fisher Yates.

```
data Suit = Clubs | Diamonds | Hearts | Spades deriving (Show, Eq, Enum)

type Rank = Int

type Card = (Rank, Suit)

main :: IO ()
main = do ...
```

Exercițiu 3

```
data Op = Plus | Mult deriving (Show, Eq)

data Elem = Number Int | Operator Op deriving (Show, Eq)

type RPNExp = [Elem]
```

Structura de date de mai sus poate fi folosită pentru a memora o expresie în `RPN` (reverse Polish notation).

Spre exemplu, expresia $3 * 4 + 5$ se reprezintă ca

$$[\texttt{Number 3}, \texttt{Number 4}, \texttt{Operator Mult}, \texttt{Number 5}, \texttt{Operator Plus}].$$

Expresiile în `RPN` se pot evalua foarte simplu cu următorul algoritm:

- folosim o stivă de numere, inițial vidă.
- citim expresia de la stânga la dreapta și:
  - dacă întâlnim un număr, îl adăugăm la stivă;
  - dacă întâlnim un operator, extragem două numere de pe stivă, calculăm rezultatul, și îl punem pe stivă.

Spre exemplu, pentru expresia de mai sus, stiva va avea, pe rând, următoarele valori:

- `[]`
- `[3]`
- `[4]`
- `[12]`
- `[12, 5]`
- `[17]`

Implementați algoritmul de mai sus folosind monada `State`.