Laboratorul 7: I/O, Interpretor

```
module Lab7 where
import Data.Char (toUpper)
import Data.List.Split (splitOn)
```

ATENTIE Fișierul `lab7literate.lhs` are extensia `.lhs` și este scris folosind "Literate Haskell" Fișierele `lab7.hs` și `lab7.pdf` sunt generate automat din fișierul `.lhs`

Puteti lucra direct în acest fișier (atât `ghci` cât și `ghc` înțeleg formatul literate Haskell) sau puteti lucra ca și până acum folosind fișierul generat `lab7.hs`.

Dacă hotărâți să folosiți fișierul `.lhs`, trebuie să respectați regula că liniile de cod incep cu `>`.

În acest laborator vom exersa concepte prezentate în cursurile 5 și 6.

Operații de intrare/ieșire în Haskell

Vom începe prin a exersa operațiile de citire și scriere.

Exemplul 1

Citirea de la tastatura a unui șir și afișarea rezultatului obținut după prelucrare.

```
prelStr strin = map toUpper strin

ioString = do
           strin <- getLine
           putStrLn $ "Intrare\n" ++ strin
           let  strout = prelStr strin
           putStrLn $ "Iesire\n" ++ strout
```

Exemplul 2

Citirea de la tastatura a unui număr și afișarea rezultatului obținut după prelucrare.

```
prelNo noin = sqrt noin
```

```
ioNumber = do
           noin <- readLn :: IO Double
           putStrLn $ "Intrare\n" ++ (show noin)
           let  noout = prelNo noin
           putStrLn $ "Iesire"
           print noout
```

Observati utilizarea functiilor “readLn”, “show” si “print”.

Exemplul 3

Citirea din fișier de intrare și afișarea rezultatului într-un fișier de ieșire.

```
inoutFile = do
              sin <- readFile "Input.txt"
              putStrLn $ "Intrare\n" ++ sin
              let sout = prelStr sin
              putStrLn $ "Iesire\n" ++ sout
              writeFile "Output.txt" sout
```

Atenție! Funcția `readFile` întoarce un rezultat de tipul `IO String`.

Exercițiul 1

Scrieți un program care citește de la tastatură un număr n și o secvență de n persoane, pentru fiecare persoană citind numele si varsta. Programul trebuie sa afiseze persoana (sau persoanele) cu varsta cea mai mare. Presupunem ca varsta este exprimata printr-un `Int`.

Exemplu de intrare:

```
3
Ion Ionel Ionescu
70
Gica Petrescu
99
Mustafa ben Muhamad
7
```

Exemplu de iesire:

```
Cel mai in varsta este Gica Petrescu (99 ani).
```

Exercițiul 2

Aceeași cerință ca mai sus, dar datele se citesc dintr-un fișier de intrare, în care fiecare linie conține informația asociată unei persoane, numele si varsta fiind separate prin vigulă (vedeți fișierul `ex2.in`).

Indicație: pentru a separa numele de varsta puteti folosi funcția `splitOn` din modulul `Data.List.Split`.

Definirea unui limbaj și a semanticii denotaționale asociate în Haskell

În cele ce urmează vom lucra cu interpretoarele definite în Cursul 6.

Exercițiul 3

Fișierul `minicalc.hs` conține limbajul unui minicalculator, împreună cu semantica lui denotațională, așa cum afost definit la curs. Observăm că un program este o expresie de tip `Prog` iar rezultatul execuției se obține apelând `prog val`.

3.1) Scrieți mai multe programe si rulați-le pentru a vă familiariza cu sintaxa.

3.2) Adăugați operația de înmulțire pentru expresii, cu precedență mai mare decât a operației de adunare. Definiți semantica operației de înmulțire.

3.3) Modificați sintaxa și semantica astfel:

- adăugați un nou constructor de tip pentru `Instr` astfel:

  ```
  data Instr  = Off | Expr  :> Instr |  Expr :>> Instr
  ```

- semantica este următoarea: pentru `Expr :> Instr` valoarea expresiei este introdusa în memorie, iar pentru `Expr :>> Instr` valoarea expresiei nu este introdusa în memorie. Astfel

  `On ((V 3) :> (( Mem :+ (V 5)) :> Off))` va afisa `[3,8]`

  `On ((V 3) :>> (( Mem :+ (V 5)) :> Off))` va afisa `[3,5]`

3.4) Modificați sintaxa și semantica astfel încât să existe două celule memorie `Mem1` și `Mem2`. Pentru fiecare celulă de memorie adăugați o instrucțiune care pune valoarea ultimei expresii în celula respectivă, dar păstrați și posibilitatea de a evalua expresiile fără a modifica memoria.

Exercițiul 4

Fișierul microHaskell.hs conține un mini-limbaj funcțional, împreună cu semantica lui denotațională, așa cum afost definit la curs. Definim comanda:

```
run :: Hask -> String
run pg = showV (hEval pg [])
```

Astfel, `run pgm` va întoarce rezultatul evaluării (rulării) programului `pgm`.

4.1) Scrieți mai multe programe si rulați-le pentru a vă familiariza cu sintaxa.

4.2) Adăugați operația de înmulțire pentru expresii, cu precedență mai mare decât a operației de adunare. Definiți semantica operației de înmulțire.

4.3) Folosind funcția `error`, înlocuiți acolo unde este posibil valoarea `VError` cu o eroare care să precizeze motivul apariției erorii.

4.4) Adăugați expresia `HLet Name Hask Hask` ca alternativă în definirea tipului `Hask`. Semantica acestei expresii este cea uzuală: `HLet x ex e` va evalua `e` într-un mediu în care `x` are valoarea lui `ex` în mediul curent. De exemplu, dacă definim

h1 = HLet "x" (HLit 3) ((HLit 4) :+: HVar "x")

atunci `run h1` va întoarce "7".