Laboratorul 6: Tipuri de date, logică, I/O

```
module Lab6 where
import Data.List (nub)
import Data.Maybe (fromJust)
```

ATENTIE Fișierul `lab6literate.lhs` are extensia `.lhs` și este scris folosind "Literate Haskell" Fișierele `lab6.hs` și `lab6.pdf` sunt generate automat din fișierul `.lhs`

Puteti lucra direct în acest fisier (atât `ghci` cât și `ghc` înțeleg formatul literate Haskell) sau puteti lucra ca și până acum folosind fișierul generat `lab6.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 4 și 5.

Încălzire

Vom începe prin a scrie câteva funcții definite folosind tipul de date `Fruct`:

```
data Fruct
  = Mar String Bool
  | Portocala String Int
```

O expresie de tipul `Fruct` este fie un `Mar String Bool` sau o `Portocala String Int`. Vom folosi un `String` pentru a indica soiul de mere sau portocale, un `Bool` pentru a indica dacă mărul are viermi și un `Int` pentru a exprima numărul de felii dintr-o portocală. De exemplu:

```
ionatanFaraVierme = Mar "Ionatan" False
goldenCuVierme = Mar "Golden Delicious" True
portocalaSicilia10 = Portocala "Sanguinello" 10
```

Exercițiul 0

Scrieți o funcție

```
ePortocalaDeSicilia :: Fruct -> Bool
ePortocalaDeSicilia = undefined
```

care indică dacă un fruct este o portocală de Sicilia sau nu. Soiurile de portocale din Sicilia sunt `Tarocco`, `Moro` și `Sanguinello`. De exemplu,

```
test_ePortocalaDeSicilia1 =
    ePortocalaDeSicilia (Portocala "Moro" 12) == True
test_ePortocalaDeSicilia2 =
    ePortocalaDeSicilia (Mar "Ionatan" True) == False
```

Logică propozițională

În restul acestui laborator vom implementa funcții pentru a lucra cu logică propozițională în Haskell. Fie dată următoarea definiție:

```
type Nume = String
data Prop
  = Var Nume
  | F
  | T
  | Not Prop
  | Prop :|: Prop
  | Prop :&: Prop
  deriving (Eq, Read)
infixr 2 :|:
infixr 3 :&:
```

Tipul `Prop` este o reprezentare a formulelor propoziționale. Variabilele propoziționale, precum `p` și `q` pot fi reprezentate va `Var "p"` și `Var "q"`. În plus, constantele booleene `F` și `T` reprezintă `false` și `true`, operatorul unar `Not` reprezintă negația ($\neg$; a nu se confunda cu funcția `not :: Bool -> Bool`) și operatorii (infix) binari `:|:` și `:&:`reprezintă disjuncția ($\vee$) și conjuncția ($\wedge$).

Exercițiul 1

Scrieți următoarele formule ca expresii de tip `Prop`, denumindu-le `p1`, `p2`, `p3`.

1. $(P \vee Q) \wedge (P \wedge Q)$

```
p1 :: Prop
p1 = undefined
```

2. $(P \vee Q) \wedge (\neg P \wedge \neg Q)$

```
p2 :: Prop
p2 = undefined
```

3. $(P \wedge (Q \vee R)) \wedge ((\neg P \vee \neg Q) \wedge (\neg P \vee \neg R))$

```
p3 :: Prop
p3 = undefined
```

Exercițiul 2

Faceți tipul `Prop` instanță a clasei de tipuri `Show`, înlocuind conectivele `Not`, `:|:` și `:&:` cu `~`, `\/` și `/\` și folosind direct numele variabilelor în loc de construcția `Var nume`.

```
instance Show Prop where
  show = undefined

test_ShowProp :: Bool
test_ShowProp =
    show (Not (Var "P") :&: Var "Q") == "((~P)/\Q)"
```

Exercițiul 2' (opțional)

Schimbați definiția lui `show` astfel încât parantezele să fie puse doar atunci când sunt strict necesare. Pentru aceasta, obsevați că o subexpresie a unui operator trebuie pusă în paranteze doar dacă precedența sa este mai mică decât cea a operatorului. Asfel, întrucât precedența lui `Not` este cea a aplicației iar precedențele lui `:&:` și `:|:` sunt 3 și respectiv 2: 1. Expresia de sub `Not` trebuie pusă în paranteze doar dacă are la vârf `:|:` sau `:&:` 2. O subexpresie a lui `:&:` trebuie pusă în paranteze doar dacă are la vârf `:|:`

Evaluarea expresiilor logice

Pentru a putea evalua o expresie logică vom considera un mediu de evaluare care asociază valori `Bool` variabilelor propoziționale:

```
type Env = [(Nume, Bool)]
```

Tipul `Env` este o listă de atribuiri de valori de adevăr pentru (numele) variabilelor propoziționale.

Pentru a obține valoarea asociată unui `Nume` în `Env`, putem folosi funcția predefinită `lookup :: Eq a => a -> [(a,b)] -> Maybe b`.

Deși nu foarte elegant, pentru a simplifica exercițiile de mai jos, vom definit o variantă a funcției `lookup` care generează o eroare dacă valoarea nu este găsită.

```
impureLookup :: Eq a => a -> [(a,b)] -> b
impureLookup a = fromJust . lookup a
```

O soluție mai elegantă ar fi să reprezentăm toate funcțiile ca fiind parțiale (rezultat de tip `Maybe`) și să folosim faptul că `Maybe` este monadă.

Exercițiul 3

Definiți o funcție `eval` care dat fiind o expresie logică și un mediu de evaluare, calculează valoarea de adevăr a expresiei.

```haskell
eval :: Prop -> Env -> Bool
eval = undefined

test_eval = eval  (Var "P" :|: Var "Q") [("P", True), ("Q", False)] == True
```

Satisfiabilitate

O formulă în logica propozițională este *satisfiabilă* dacă există o atribuire de valori de adevăr pentru variabilele propoziționale din formulă pentru care aceasta se evaluează la `True`.

Pentru a verifica dacă o formulă este satisfiabilă vom genera toate atribuirile posibile de valori de adevăr și vom testa dacă formula se evaluează la `True` pentru vreuna dintre ele.

Exercițiul 4

Definiți o funcție `variabile` care colectează lista tuturor variabilelor dintr-o formulă. *Indicație*: folosiți funcția `nub`.

```haskell
variabile :: Prop -> [Nume]
variabile = undefined

test_variabile =
  variabile (Not (Var "P") :&: Var "Q") == ["P", "Q"]
```

Exercițiul 5

Dată fiind o listă de nume, definiți toate atribuirile de valori de adevăr posibile pentru ea.

```haskell
envs :: [Nume] -> [[(Nume, Bool)]]
envs = undefined

test_envs =
    envs ["P", "Q"]
    ==
    [ [ ("P",False)
      , ("Q",False)
      ]
    , [ ("P",False)
      , ("Q",True)
      ]
    , [ ("P",True)
      , ("Q",False)
      ]
    , [ ("P",True)
```

```
      , ("Q",True)
      ]
    ]
```

Exercițiul 6

Definiți o funcție `satisfiabila` care dată fiind o Propoziție verifică dacă aceasta este satisfiabilă. Puteți folosi rezultatele de la exercițiile 4 și 5.

```
satisfiabila :: Prop -> Bool
satisfiabila = undefined

test_satisfiabila1 = satisfiabila (Not (Var "P") :&: Var "Q") == True
test_satisfiabila2 = satisfiabila (Not (Var "P") :&: Var "P") == False
```

Exercițiul 7

O propoziție este validă dacă se evaluează la `True` pentru orice interpretare a varibilelor. O forumare echivalenta este aceea că o propoziție este validă dacă negația ei este nesatisfiabilă. Definiți o funcție `valida` care verifică dacă o propoziție este validă.

```
valida :: Prop -> Bool
valida = undefined

test_valida1 = valida (Not (Var "P") :&: Var "Q") == False
test_valida2 = valida (Not (Var "P") :|: Var "P") == True
```

Exercițiul 8

Definiți o funcție `tabelaAdevar` care afișează tabela de adevăr corespunzătoare unei expresii date.

```
tabelaAdevar :: Prop -> IO ()
tabelaAdevar = undefined
```

Indicație: folosiți exercițiile 4 și 5. De exemplu:

```
*Main> tabelaAdevar ((Var "P" :&: Not (Var "Q")) :&: (Var "Q" :|: Var "P"))
P Q | ((P/\(~Q))/\(Q\/P))
- - | ---------------
F F |        F
F T |        F
T F |        T
T T |        F
```

Implicație și echivalență

Exercițiul 9

Extindeți tipul de date `Prop` și funcțiile definite până acum pentru a include conectivele logice `->` (implicația) și `<->` (echivalența), folosind constructorii `:->:` și `:<->:`. După ce le implementați, tabelele de adevăr pentru ele trebuie să arate astfel:

```
*Main> table (Var "P" :->: Var "Q")
P Q | (P->Q)
- - | -----
F F |   T
F T |   T
T F |   F
T T |   T
```

```
*Main> table (Var "P" :<->: Var "Q")
P Q | (P<->Q)
- - | ------
F F |    T
F T |    F
T F |    F
T T |    T
```

Exercițiul 10

Două propoziții sunt echivalente dacă au mereu aceeași valoare de adevăr, indiferent de valorile variabilelor propoziționale. Scrieți o funcție care verifică dacă două propoziții sunt echivalente.

```haskell
echivalenta :: Prop -> Prop -> Bool
echivalenta = undefined

test_echivalenta1 =
  True
  ==
  (Var "P" :&: Var "Q") `echivalenta` (Not (Not (Var "P") :|: Not (Var "Q")))
test_echivalenta2 =
  False
  ==
  (Var "P") `echivalenta` (Var "Q")
test_echivalenta3 =
  True
  ==
  (Var "R" :|: Not (Var "R")) `echivalenta` (Var "Q" :|: Not (Var "Q"))
```