

Laboratorul 8: Logică propozițională

În acest 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
infixr 2 :|:
infixr 3 :&:
```

Tipul `Prop` este o reprezentare a formulelor propoziționale. Variabilele propoziționale, precum `p` și `q` pot fi reprezentate ca `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 = (Var "P" :|: Var "Q") :&: (Var "P" :&: Var "Q")
```

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, observați că o subexpresie a unui operator trebuie pusă în paranteze doar dacă precedența sa este mai mică decât cea a operatorului. Astfel, 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 defini 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 controlăm propagarea erorilor.

Exercițiul 3

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

```
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`.

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

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

Exercițiul 5

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

```
envs :: [Nume] -> [Env]
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 variabilelor. O formulare echivalentă 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

Toate evaluările unei propoziții pot fi reprezentate folosind tabelul de adevăr:

P	Q	(P&(~Q))
-	-	-----
F	F	F
F	T	F
T	F	T
T	T	F

Definiți o funcție **tabelAdevar** care afișează tabela de adevăr corespunzătoare unei expresii date astfel încât fiecare linie din tabelul de adevăr să fie separată de caracterul `\n`.

```

tabelAdevar :: Prop -> String
tabelAdevar = undefined

```

Indicație: folosiți exercițiile 4 și 5.

De exemplu:

```
*Main> tabelAdevar ((Var "P") :&: (Not (Var "Q")))
"P Q (P&(~Q))\nF F F\nF T F\nT F T\nT T F\n"
```

Observatie: putem afisa pe linii tabelul folosind functie `putStrLn`

```
*Main>putStrLn$ tabelAdevar ((Var "P") :&: (Not (Var "Q")))
P Q (P&(~Q))
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> tabelaAdevar (Var "P" :->: Var "Q")
P Q | (P->Q)
- - | -----
F F |      T
F T |      T
T F |      F
T T |      T

*Main> tabelaAdevar (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.

```
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"))

```