# Laboratorul 7: Testare folosind QuickCheck

În Haskell avem la dispoziție o librărie care, în anumite situații, generează teste automate. Modificați fișierul lab10.hs astfel:

1. Importați modulul `Test.QuickCheck`, i.e. adăugați la începutul fișierului `import Test.QuickCheck`. Dacă acest modul nu este instalat, puteți instala folosind următoarele comenzi direct în terminal (cmd, powershell):

```
cabal update
cabal install QuickCheck
```

Dacă folosiți Windows și `chocolately`, e posibil să trebuiască să faceți configurare locală în directorul în care veți folosi `QuickCheck`, astfel:

```
c:\Users\...\ex1haskell> cabal install --lib --package-env . QuickCheck
```

Pentru mai multe detalii consultați răspunsul de pe Stack Overflow.

2. Definiți următoarele funcții care calculează dublul, triplul, respectiv, de cinci ori numărul dat ca parametru.

```
double :: Int -> Int
double = undefined
triple :: Int -> Int
triple = undefined
penta :: Int -> Int
penta = undefined
```

3. Observați în fișier următoarea funcție test:

```
test x = (double x + triple x) == (penta x)
```

4. Ce tip are funcția `test`?
5. În interpretor evaluați

```
*Main> quickCheck test
```

și observați rezultatul.

6. Scrieți un alt test care să verifice o proprietate falsa, verificați cu `quickCheck` și observați rezultatul.
7. Scrieți o funcție

```
myLookUp :: Int -> [(Int,String)]-> Maybe String
myLookUp = undefined
```

care caută un element întreg într-o listă de perechi cheie-valoare și întoarce valoarea gasită folosind un răspuns de tip `Maybe String`.

Scrieți un test

```
testLookUp :: Int -> [(Int,String)] -> Bool
testLookUp = undefined
```

care verifică faptul că funcția `myLookUp` are aceleași rezultate ca funcția `lookUp` predefinită.

### QuickCheck cu constrângeri

Să încercăm să testăm că `myLookUp` este echivalentă cu `lookUp` predefinită doar pentru chei pozitive și divizibile cu 5.

```
-- testLookUpCond :: Int -> [(Int,String)] -> Property
-- testLookUpCond n list = n > 0 && n `div` 5 == 0 ==> testLookUp n list
```

Observați faptul că `testLookUpCond` are ca rezultat `Property`. Constrângerea din stânga „implicației” `==>` selecționeaza din valorile de intrare generate doar pe acelea care satisfac condiția dată. Evaluați în interpretor `quickCheck testLookUpCond` și observați rezultatul.

8. (a) Scrieți o funcție `myLookUp'` cu aceeași signatură ca `myLookup` care atunci când găsește valoarea, capitalizează prima literă.

   (b) Scrieți un predicat care testează că `myLookUp'` este echivalentă cu `lookUp` pentru listele care conțin doar valori care încep cu majusculă. Verificați-l folosind `quickCheck`.

### Testare pentru tipuri de date algebrice

Definiți o instanță a clasei `Arbitrary` pentru tipul de date `ElemIS` (instanțe similare în **cursul 8**)

```
data ElemIS = I Int | S String
     deriving (Show,Eq)
```

9. Definiți o funcție `myLookUpElem` care funcționează similar cu `lookUp`, doar ca funcționează pentru chei de tip întreg și valori de tip ElemIS.

```
myLookUpElem :: Int -> [(Int,ElemIS)]-> Maybe ElemIS
myLookUpElem = undefined
```

Scrieți un test

```
testLookUpElem :: Int -> [(Int,ElemIS)] -> Bool
testLookUpElem = undefined
```

Și rulați în consolă `quickCheck testLookUpElem`.