FMI, Info, Anul III, 2018-2019
Programare declarativă

Laborator 2
Liste în Haskell
Definiţii prin comprehensiune şi recursie

1 Recursie

Una dintre diferențele dintre programarea declarativă și cea imperativă este modalitatea de abordare a problemei iterării: în timp ce in programarea imperativă acesta este rezolvată prin bucle (**while**, **for**, ...), în programarea declarativă rezolvarea iterării se face prin conceptul de recursie.
Un avantaj al recursiei față de bucle este acela că usurează sarcina de scriere și verificare a corectitudinii programelor prin raționamente de tip inductiv: construiește rezultatul pe baza rezultatelor unor subprobleme mai simple (aceeași problemă, dar pe o dimensiune mai mică a datelor).

Exemplu: Fibonacci Un foarte simplu exemplu de recursie este acela al calculării unui element de index dat din secvența numerelor Fibonacci, definită recursiv de:

$$F_n = \begin{cases} n & \text{dacă } n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{dacă } n > 1 \end{cases}$$

Putem transcrie această definiție direct in Haskell:

```
fibonacciCazuri :: Integer -> Integer
fibonacciCazuri n
    | n < 2     = n
    | otherwise = fibonacciCazuri (n - 1) + fibonacciCazuri (n - 2)
```

Alternativ, putem folosi o definiție în stil ecuațional (cu șabloane):

```
fibonacciEcuational :: Integer -> Integer
fibonacciEcuational 0 = 0
fibonacciEcuational 1 = 1
fibonacciEcuational n =
    fibonacciEcuational (n - 1) + fibonacciEcuational (n - 2)
```

Fibonacci liniar O problemă cu definiția de mai sus este aceea că este timpul ei de execuție este exponențial. Motivul este acela că rezultatul este compus din rezultatele a 2 subprobleme de mărime aproximativ egală cu cea inițială.
Dar, deoarece recursia depinde doar de precedentele 2 valori, o putem simplifica cu ajutorul unei funcții care calculează recursiv perechea (F_{n-1}, F_n).

(L2.1) [**Fibonacci liniar**] Completați definiția funcției fibonacciPereche
Observație 1. Folosiți principiul de inducție: ne bazăm pe faptul ca fibonacciPereche (n−1) va calcula perechea (F_{n-2}, F_{n-1}) și o folosim pe aceasta pentru a calcula perechea (F_{n-1}, F_n).
Observație 2. Recursia este liniară *doar dacă* expresia care reprezintă apelul recursiv apare o singură dată. Folosiți **let**, **case**, sau **where** pentru a vă asigura de acest lucru.

QuickCheck cu constrângeri Să incercăm sâ testâm că fibonacciLiniar este echivalentă cu una din implementările precedente.
Un test că cele două funcții sunt echivalente pentru toate valorile de intrare ar fi:

```
prop_fibonacci :: Integer -> Bool
prop_fibonacci n = fibonacciEcuational n == fibonacciLiniar n
```

Totuși, funcțiile nu sunt definite decât pentru valori naturale. Putem specifica acest lucru în proprietate, astfel:

```
prop_fibonacci :: Integer -> Property
prop_fibonacci n =
    n >= 0 ==> fibonacciEcuational n == fibonacciLiniar n
```

Observați faptul că prop_fibonacci are ca rezultat Property. Constrângerea din stânga „implicației” `==>` selcționează din valorile de intrare generate doar pe acelea care satisfac condiția dată.
Totuși, dacă încercăm să testăm proprietatea în forma ei actuală, vom vedea că testarea durează foarte mult. Acest lucru e din cauză că fibonacciEcuational este exponențial.

(L2.2) [**QuickCheck cu constrângeri**] Modificați prop_fibonacci pentru a constrânge datele de intrare și superior astfel încât testarea să se termine în timp rezonabil.

2 Recursie peste liste

Listele sunt unul dintre cele mai simple exemple de structuri de date definite inductiv. O listă este fie **vidă**, fie **construită** prin adăugarea unui element (**head**) unei *liste* existente (**tail**).
Listele fiind definite inductiv, recursia este o modalitate naturală de a le traversa.

Exemplu Dată fiind o listă de numere întregi, să se scrie o funcție semiPare care elimină numerele impare și le injumătățește pe cele pare. De exemplu:

```
semiPare [0,2,1,7,8,56,17,18] == [0,1,4,28,9]
```

Prima implementare propusă este realizabilă in orice limbaj, folosind testul **null**, și „destructorii" **head** și **tail**.

```
semiPareRecDestr :: [Int] -> [Int]
semiPareRecDestr l
  | null l    = l
  | even h    = h `div` 2 : t'
  | otherwise = t'
  where
    h = head l
    t = tail l
    t' = semiPareRecDestr t
```

A doua implementare (preferată) folosește șabloane peste constructorul de listă : pentru a descompune lista:

```
semiPareRecEq :: [Int] -> [Int]
semiPareRecEq [] = []
semiPareRecEq (h:t)
  | even h    = h`div` 2 : t'
  | otherwise = t'
  where t' = semiPareRecEq t
```

(L2.3) [**În interval**] Scrieți o funcție inInterval care date fiind limita inferioară și cea superioară (întregi) a unui interval închis și o listă de numere întregi, calculează lista numerelor din listă care aparțin intervalului. De exemplu:

```
inInterval 5 10 [1..15] == [5,6,7,8,9,10]
```

3 Liste definite prin comprehensiune

Haskell permite definirea unei liste prin selectarea și transformarea elementelor din alte liste sursă, folosind o sintaxă asemănătoare definirii mulțimilor matematice:
[expresie | selectori, legari, filtrari]
unde:

selectori una sau mai multe construcții de forma pattern <− elista (separate prin virgulă) unde elista este o expresie reprezentând o listă iar pattern este un șablon pentru elementele listei elista

legari zero sau mai multe expresii (separate prin virgulă) de forma **let** pattern = expresie folosind la legarea corespunzătoare a variabilelor din pattern cu valoarea expresie.

filtrari zero sau mai multe expresii de tip **Bool** (separate prin virgulă) folosite la eliminarea instanțelor selectate pentru care condiția e falsă

expresie expresie descriind elementele listei rezultat

Exemplu Iată cum arată o posibilă implementare a funcției semiPare folosind descrieri de liste:

```
semiPareComp :: [Int] -> [Int]
semiPareComp l = [ x `div` 2 | x <- l, even x ]
```

(L2.4) [**În interval, din nou**] Scrieți o funcție inIntervalComp care folosește descrieri de liste pentru a implementa aceeași funționalitate ca funcția inInterval definită mai sus. Scrieți și o proprietate care descrie echivalența dintre cele două definiții și folosiți quickCheck pentru a o testa.

(L2.5) [**Numarăm pozitive**] Scrieți o funcție pozitive care numără câte numere strict pozitive sunt într-o listă dată ca argument. De exemplu:

```
pozitive [0,1,-3,-2,8,-1,6] == 3
```

- Folosiți doar recursie. Denumiți funcția pozitiveRec
- Folosiți descrieri de liste. Denumiți funcția pozitiveComp Nu puteți folosi recursie, dar veți avea nevoie de o funcție de agregare. (Consultați modulul Data.List)
- Definiți o proprietate pentru a verifica echivalența celor două implementări și testați-o folosind quickCheck
- De ce nu e posibil să scriem pozitiveComp doar folosind descrieri de liste?

(L2.6) [**Poziții**] Scrieți o funcție pozitiiImpare care dată fiind o listă de numere calculează lista pozițiilor elementelor impare din lista originală. De exemplu:

```
pozitiiImpare [0,1,-3,-2,8,-1,6,1] == [1,2,5,7]
```

- Folosiți doar recursie. Denumiți funcția pozitiiImpareRec

  Indicație: folosiți o funcție ajutătoare, cu un argument în plus reprezentând poziția curentă din listă.

- Folosiți descrieri de liste. Denumiți funcția pozitiiImpareComp.

  Indicație: folosiți funcția **zip** pentru a asocia poziții elementelor listei (puteți căuta exemplu în curs).

- Definiți o proprietate pentru a verifica echivalența celor două implementări și testați-o folosind quickCheck

4 Direcții și drumuri

În cadrul acestui exercițiu vom modela o rețea de tip grilă 2D cu coordonate întregi. Vom defini drumuri în această rețea și proprietăți ale lor.

(L2.7) [**Directie**] Definiți un tip de date Directie reprezentând cele patru puncte cardinale: Nord, Est, Sud, Vest. (Folosiți **data** și **deriving** (**Eq**, **Show**)).

(L2.8) [**Punct**] Definiți tipul de date Punct reprezentând perechi de coordonate pe grilă. (Folosiți **type**)

(L2.9) [**Origine**] Definiți un punct, numit origine care reprezintă originea grilei.

(L2.10) [**Drum**] Definiți tipul de date Drum care reprezintă un drum pe grilă ca o listă de direcții (folosiți **type**).

(L2.11) [**Mișcare**] Definiți o funcție miscare :: Punct −> Drum −> Punct care calculează punctul de pe grilă în care se ajunge din primul argument (Punct) după urmarea instrucțiunilor specificate de al doilea argument (Drum). Folosim convenția (uzualâ) că mergând înspre nord creștem a doua coordonată iar mergând înspre est creștem prima coordonată.
Sugestie: Definiți o funcție auxiliară pentru mutarea după o singură direcție.

(L2.12) [**Egalitate**] Definiți o funcție eqDrum :: Drum −> Drum −> **Bool** care testează dacâ două rute sunt echivalente în sensul că ajung în același punct dacă pornesc din origine.

Material suplimentar

- Citiţi capitolul *Recusion* din

  M. Lipovaca, Learn You a Haskell for Great Good!

  http://learnyouahaskell.com/recursion

- Efectuaţi exerciţiile din laboratorul suplimentar (continuare a laboratorului suplimentar de săptămîna trecută).