FMI, Info, Anul III, 2019-2020
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.

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
```

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 ]
```

(L2.2) [În interval] Scrieți o funcție 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]

inInterval 5 10 [1,3,5,2,8,-1] = [5,8]
```

- Folosiți doar recursie. Denumiți funcția inIntervalRec
- Folosiți descrieri de liste. Denumiți funcția inIntervalComp

(L2.3) [**Numarăm pozitive**] Scrieți o funcție 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)
- De ce nu e posibil să scriem pozitiveComp doar folosind descrieri de liste?

(L2.4) [**Poziții**] Scrieți o funcție 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).

(L2.5) [**MultDigit**] Scrieți o funcție care calculează produsul tuturor cifrelor care apar în șirul de caractere dat ca intrare. Dacă nu sunt cifre în șir, răspunsul funcției trebuie să fie 1. De exemplu:

```
multDigits "The time is 4:25" == 40

multDigits "No digits here!" == 1
```

- Folosiți doar recursie. Denumiți funcția multDigitsRec
- Folosiți descrieri de liste. Denumiți funcția multDigitsComp

  Indicație: Veți avea nevoie de funcția **isDigit** care verifică dacă un caracter e cifră și funcția **digitToInt** care transformă un caracter in cifră.

(L2.6) [**Discount**] Scrieți o funcție care pentru o listă de valori (reprezentând niște prețuri) aplică un discount de 25% acelor valori si pastrează în listă valorile reduse care sunt mai mici decât 200. De exemplu:

```
discount [150, 300, 250, 200, 450, 100] == [112.5, 187.5, 150.0, 75.0
    ]
```

- Folosiți doar recursie. Denumiți funcția discountRec
- Folosiți descrieri de liste. Denumiți funcția discountComp

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).