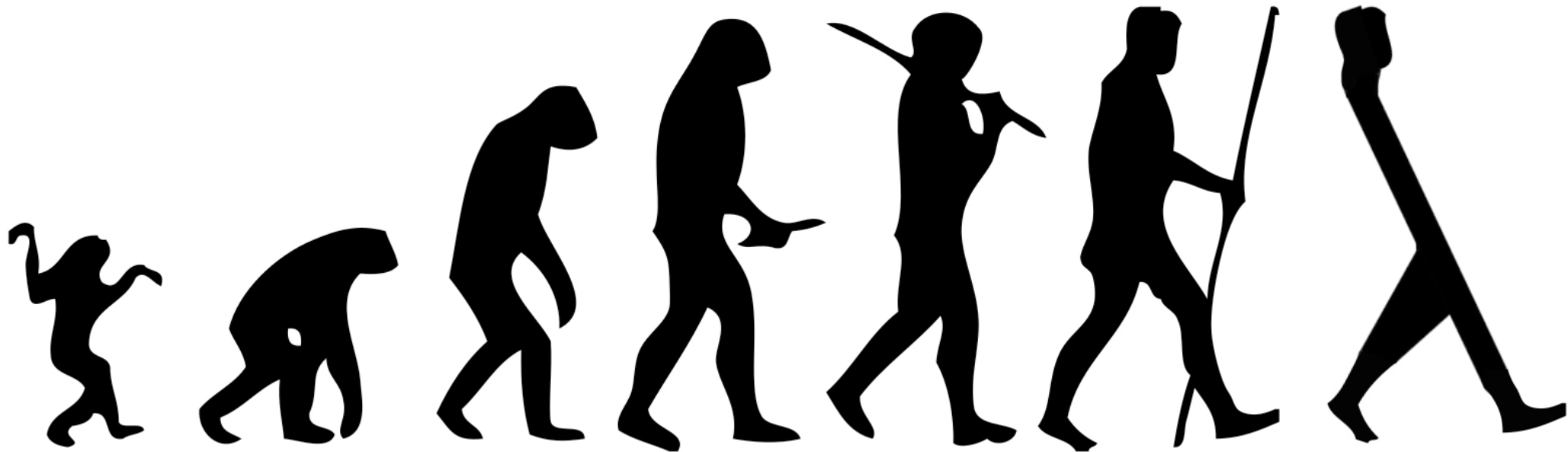


PARADIGME DE PROGRAMARE

Curs 6

Limbajul Haskell.

Programare funcțională în Haskell



Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare leneșă

Sintaxa Haskell

- **Notăție infixată pentru operatori**

`1 + 2`, `a < 5`

- **Notăție prefixată pentru funcții**

`filter odd [1 .. 7]`, `f 2 + f 5`

- **Parantezele se folosesc pentru controlul priorității (nu pentru aplicația de funcție)**

`f (1 + 2)`, `f (g x)`

`(f 1) + 2` este echivalent cu `f 1 + 2`, întrucât aplicația de funcție are cea mai mare prioritate

`(f g) x` este echivalent cu `f g x`, întrucât aplicația de funcție este asociativă la stânga

- **Indentarea înlocuiește controlul prin separatori ca `{ }` sau `;`**

Orice cod din **corpul** unei expresii trebuie indentat **mai la dreapta** decât începutul expresiei!

O **nouă expresie** începe pe **același nivel sau mai la stânga** față de începutul expresiei anterioare!

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare leneșă

TDA-ul Pereche

Constructori de bază

`(,)` : $T_1 \times T_2 \rightarrow$ Pereche // creează o pereche între orice 2 argumente

Operatori

`fst` : Pereche $\rightarrow T_1$ // extrage prima valoare din pereche

`snd` : Pereche $\rightarrow T_2$ // extrage a doua valoare din pereche

Exemple

```
(1, "unu")
```

```
a = (('a', 2), "a2")
```

```
fst a
```

```
snd (fst a)
```

TDA-ul Pereche

Constructori de bază

`(,)` : $T_1 \times T_2 \rightarrow$ Pereche // creează o pereche între orice 2 argumente

Operatori

`fst` : Pereche $\rightarrow T_1$ // extrage prima valoare din pereche

`snd` : Pereche $\rightarrow T_2$ // extrage a doua valoare din pereche

Exemple

```
(1, "unu") -- (1, "unu")
```

```
a = (('a', 2), "a2")
```

```
fst a -- ('a', 2)
```

```
snd (fst a) -- 2
```

TDA-ul Listă

Constructori (de bază și nu numai)

[] : -> Listă

// creează o listă vidă

: : T x Listă -> Listă

// creează o listă prin adăugarea unei valori la începutul unei liste

[,,..,] : T x .. T -> Listă

// creează o listă din toate argumentele sale (de același tip)

Operatori

head : Listă -> T

tail : Listă -> Listă

null : Listă -> Bool

length : Listă -> Nat

++ : Listă x Listă -> Listă

Exemple

head [[2, 4], [6], [5]]

tail (2:3:[4, 5])

null [[]]

length [[]]

[1] ++ [1, 2, 3] ++ [4, 5]

TDA-ul Listă

Constructori (de bază și nu numai)

[] : -> Listă

// creează o listă vidă

: : T x Listă -> Listă

// creează o listă prin adăugarea unei valori la începutul unei liste

[,,..,] : T x .. T -> Listă

// creează o listă din toate argumentele sale (de același tip)

Operatori

head : Listă -> T

tail : Listă -> Listă

null : Listă -> Bool

length : Listă -> Nat

++ : Listă x Listă -> Listă

Exemple

```
head [[2,4],[6],[5]]      -- [2,4]
tail (2:3:[4,5])         -- [3,4,5]
null [[]]                -- False
length [[]]              -- 1
[1] ++ [1,2,3] ++ [4,5] -- [1,1,2,3,4,5]
```

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- **Funcții**
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare leneșă

Funcții anonime în Haskell

```
\parametri -> corp
```

Exemple

```
λx.x            \x -> x
```

```
λx.λy.(x y)    \x -> \y -> x y    echivalent cu  
                 \x y -> x y        (întrucât funcțiile Haskell sunt automat curry)
```

```
(λx.x λx.y)    (\x -> x) (\x -> y)
```

Funcții cu nume în Haskell

```
f parametri = corp
```

Exemple

```
arithmeticMean = \x -> \y -> (x + y) / 2
```

echivalent cu

```
arithmeticMean = \x y -> (x + y) / 2
```

și cu

```
arithmeticMean x y = (x + y) / 2
```

```
f = arithmeticMean 3
```

```
-- se creează \y -> (3 + y) / 2
```

```
f 18
```

```
-- 10.5
```

Simularea funcțiilor uncurry

Definițiile de tipul

```
f x1 x2 ... xn = corp
```

generează funcții **curry**, care pot fi aplicate pe oricâți ($\leq n$) parametri la un moment dat.

```
f e1 e2 ... ek întoarce o nouă funcție \xk+1 ... xn -> corp[ei/xi].
```

Pentru a obține comportamentul de funcție **uncurry**, parametrul lui f trebuie să fie un tuplu:

```
f (x1, x2 ..., xn) = corp
```

Exemplu

```
arithmeticMean (x, y) = (x + y) / 2
```

```
arithmeticMean (3, 18)
```

Transformări operator – funcție

- **(op)** face transformarea **operator** → **funcție**

```
(-) 3 5 -- -2  
(||) (1<2) (5<3) -- True  
foldr (+) 0 [1..5] -- 15  
(/=) 2 2 -- False
```

- **`f`** face transformarea **funcție** → **operator**

```
5 `mod` 3 -- 2  
(div 6) `map` [1,2,3] -- [6,3,2]  
((==) 2) `filter` [1,2,3] -- [2]
```

Secțiuni (aplicare parțială a operatorilor)

- Când se dă operandul din stânga, se așteaptă operandul din dreapta

```
(5/) 2
```

```
map (2-) [0..4]
```

```
filter (2<) [0..4]
```

- Când se dă operandul din dreapta, se așteaptă operandul din stânga

```
(/5) 2
```

```
map (-2) [0..4]
```

```
map (/2) [0..4]
```

```
filter (<2) [0..4]
```

Secțiuni (aplicare parțială a operatorilor)

- Când se dă operandul din stânga, se așteaptă operandul din dreapta

```
(5/) 2          -- 2.5  
map (2-) [0..4] -- [2,1,0,-1,-2]  
filter (2<) [0..4] -- [3,4]
```

- Când se dă operandul din dreapta, se așteaptă operandul din stânga

```
(/5) 2  
map (-2) [0..4]  
map (/2) [0..4]  
filter (<2) [0..4]
```


Secțiuni (aplicare parțială a operatorilor)

- Când se dă operandul din stânga, se așteaptă operandul din dreapta

```
(5/) 2          -- 2.5
map (2-) [0..4] -- [2,1,0,-1,-2]
filter (2<) [0..4] -- [3,4]
```

- Când se dă operandul din dreapta, se așteaptă operandul din stânga

```
(/5) 2          -- 0.4
map (-2) [0..4] -- eroare, aici -2 e număr, nu funcție
map (/2) [0..4] -- [0.0,0.5,1.0,1.5,2.0]
filter (<2) [0..4] -- [0,1]
```

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare leneșă

Definirea funcțiilor prin pattern matching

Se descrie comportamentul funcțiilor în funcție de structura parametrilor – ca la scrierea axiomelor TDA-ului.

Exemple

1. `fib 0 = 0`
 2. `fib 1 = 1`
 3. `fib n = fib (n-2) + fib (n-1)`
 - 4.
 5. `sumL [] = 0`
 6. `sumL (x:xs) = x + sumL xs`
 - 7.
 8. `sumP (x,y) = x + y`
 - 9.
 10. `ordered [] = True`
 11. `ordered [x] = True`
 12. `ordered (x:xs@(y:rest)) = x <= y && ordered xs`
- permite crearea unui alias pentru valoarea următoare

La aplicare

1. `fib 0 = 0`
2. `fib 1 = 1`
3. `fib n = fib (n-2) + fib (n-1)`

`fib 3`

- Dacă argumentul se potrivește cu parametrul din primul punct (primul pattern)
 - Se folosește definiția din primul punct
 - Se ignoră definițiile următoare
- Altfel
 - Se încearcă potrivirea cu punctul următor, ș.a.m.d.

Consecință: Ordinea contează!

Pattern-uri exhaustive

Este important să specificăm comportamentul funcției pe toate valorile tipului – ca la scrierea axiomelor TDA-ului.

1. `ordered [] = True`
 2. `ordered [x] = True`
 3. `ordered (x:xs@(y:rest)) = x <= y && ordered xs`
- De aceea a trebuit să specificăm și ce se întâmplă pe lista vidă, și ce se întâmplă pe lista cu un singur element

O definiție alternativă pentru funcția `ordered`:

1. `ordered2 (x:xs@(y:rest)) = x <= y && ordered2 xs`
2. `ordered2 _ = True`

Se traduce prin „orice altceva”
Unde în definiția lui `ordered` se mai putea folosi?

Când se poate folosi pattern matching

- De fiecare dată **când se leagă variabile**

- La definirea funcțiilor
- La crearea de legări locale folosind `let` sau `where` (vom vedea)

- Pattern-urile **nu** se potrivesc și între ele

```
eq x x = True    -- ă eroare Conflicting definitions for `x'
```

```
eq _ _ = False
```

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii `.` și `$`
- Expresii condiționale
- Legare statică
- Evaluare leneșă

Operatorii . și \$

- **Operatorul .** (punct) realizează **compunere de funcții**

```
myLast = head . reverse -- myLast [1..5] =
```

```
myMin = head . sort -- myMin [2,4,1,2,3,6,2] =
```

```
myMax = myLast . sort -- myMax [2,4,1,2,3,6,2] =
```

- **Operatorul \$** (dolar) realizează **aplicație de funcție**

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
```

- $f \$ a = f a$ este interesant pentru că \$ are o **prioritate foarte mică**, astfel încât ambele părți vor fi evaluate înainte să se realizeze aplicația de funcție → evităm astfel să folosim foarte multe paranteze
- \$ este **asociativ la dreapta** și este util pentru a rescrie structuri de genul **f (g (h ... x))**, nu poate suplini orice fel de paranteze (vezi (odd . fst) mai sus)

Operatorii . și \$

- **Operatorul .** (punct) realizează **compunere de funcții**

```
myLast = head . reverse -- myLast [1..5] = 5
```

```
myMin = head . sort -- myMin [2,4,1,2,3,6,2] = 1
```

```
myMax = myLast . sort -- myMax [2,4,1,2,3,6,2] = 6
```

- **Operatorul \$** (dolar) realizează **aplicație de funcție**

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
```

- $f \$ a = f a$ este interesant pentru că \$ are o **prioritate foarte mică**, astfel încât ambele părți vor fi evaluate înainte să se realizeze aplicația de funcție → evităm astfel să folosim foarte multe paranteze
- \$ este **asociativ la dreapta** și este util pentru a rescrie structuri de genul **f (g (h ... x))**, nu poate suplini orice fel de paranteze (vezi (odd . fst) mai sus)

Operatorii . și \$

- **Operatorul .** (punct) realizează **compunere de funcții**

```
myLast = head . reverse -- myLast [1..5] = 5
```

```
myMin = head . sort -- myMin [2,4,1,2,3,6,2] = 1
```

```
myMax = myLast . sort -- myMax [2,4,1,2,3,6,2] = 6
```

- **Operatorul \$** (dolar) realizează **aplicație de funcție**

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
```

```
-- [(1,2), (3,4), (5,6), (7,8)]
```

- $f \$ a = f a$ este interesant pentru că \$ are o **prioritate foarte mică**, astfel încât ambele părți vor fi evaluate înainte să se realizeze aplicația de funcție → evităm astfel să folosim foarte multe paranteze
- \$ este **asociativ la dreapta** și este util pentru a rescrie structuri de genul **f (g (h ... x))**, nu poate suplini orice fel de paranteze (vezi (odd . fst) mai sus)

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare leneșă

Condiționala **if**

if condiție **then** rezultatThen **else** rezultatElse

Exemple

```
if 2<3 then "all normal" else "what did just happen?"  -- "all normal"
```

```
uglySum l = if null l then 0  
           else head l + uglySum (tail l)
```


Observație

- Un cod Haskell elegant va folosi pattern matching sau gărzi (vom vedea) înainte de a folosi `if`

Condiționala **case**

```
case expresie of  
  pattern1 -> rezultat1  
  pattern2 -> rezultat2  
  ...  
  patternn -> rezultatn
```

Are sens atunci când nu putem folosi pattern matching sau gărzi, de exemplu aici când verificăm structura lui head matrix, mai degrabă decât pe a lui matrix



Exemplu

```
myTranspose matrix = case (head matrix) of  
  [] -> []  
  _ -> map head matrix : myTranspose (map tail matrix)
```

Gărzi

```
f parametri  
  | condiție1 = rezultat1  
  | condiție2 = rezultat2  
  ...  
  | condițien = rezultatn  
  [| otherwise = rezultatn] ← opțional
```

Exemplu

```
allEqual a b c  
  | a==b = b==c  
  | otherwise = False
```

Au sens atunci când punem condiții asupra variabilelor, mai degrabă decât să le potrivim cu o anumită structură (caz în care am folosi pattern matching)

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare leneșă

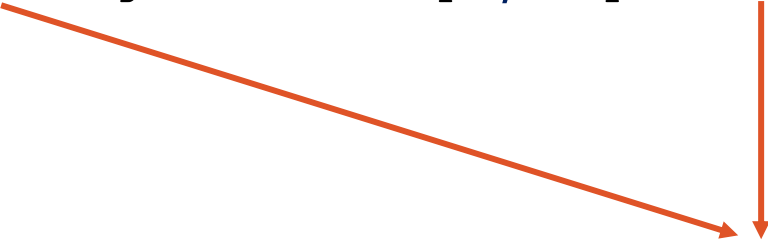
Legarea variabilelor în Haskell – statică

- Doar legare **statică**
- Expresii pentru legare locală: **let** legări **in** expr, expr **where** legări

Exemple

```
myFoldl f acc [] = acc
myFoldl f acc (x:xs) =
    let
        newAcc = f acc x
    in myFoldl f newAcc xs
```

```
myFoldr f acc [] = acc
myFoldr f acc (x:xs) = f x rightResult
    where rightResult = myFoldr f acc xs
```



Au sens pentru a spori lizibilitatea codului
sau pentru a evita apelarea repetată a
aceleiași funcții pe aceleași argumente

Limbajul Haskell – Cuprins

- Sintaxă
- Perechi și liste
- Funcții
- Pattern matching
- Operatorii . și \$
- Expresii condiționale
- Legare statică
- Evaluare leneșă

Evaluare leneșă

- Toate funcțiile sunt **nestrict**
- **Evaluare leneșă:** subexpresiile (argumentele) sunt pasate funcției fără a fi evaluate, în corpul funcției ele se vor evalua (eventual parțial) maxim o dată

Exemple

1. `f x = 2*x`

2. `g x = f 2 + f 2`

3. `h x = x*x*x`

`g 5`

-- 2 aplicări distincte ale lui f => se evaluează de 2 ori

-- `f 2 + f 2 -> 4 + f 2 -> 4 + 4 -> 8`

`h (f 2)`

-- argumentul se evaluează o dată și se folosește de 3 ori

-- `(f 2)*(f 2)*(f 2) -> 4*4*4 -> 64`

Fluxuri

- Evaluare leneșă => toate **listele sunt fluxuri** (se evaluează în măsura în care e nevoie)

Exemple

```
naturals = let loop n = n : loop (n+1) in loop 0
```

```
ones = 1 : ones
```

```
fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)
```

```
evens = filter even naturals
```

Generarea intervalelor

`[start..stop]` sau `[start..]`

`[start,next..stop]` sau `[start,next..]` `--next` dă pasul

Exemple

`[1..5]`

`[1,3..10]`

`[10,7..0]`

`[20,19.5..]`

Generarea intervalelor

`[start..stop]` sau `[start..]`

`[start,next..stop]` sau `[start,next..]` `--next` dă pasul

Exemple

`[1..5]` `-- [1,2,3,4,5]`

`[1,3..10]` `-- [1,3,5,7,9]`

`[10,7..0]` `-- [10,7,4,1]`

`[20,19.5..]` `-- lista infinită [20,19.5,19,18.5..]`

List comprehensions

```
[ expr | generatori, condiții, legări locale ]
```

Exemple

Întâi toate rezultatele pentru x=1, apoi toate pentru x=2, apoi toate pentru x=3



```
lc1 = [ (x,y,z) | x<-[1..3], y<-[1..4], x<y, let z = x+y, odd z ]
```

```
fibonacci = 0 : 1 : [ x+y | (x,y) <- zip fibonacci (tail fibonacci) ]
```

```
qsort [] = []
```

```
qsort (x:xs) =
```

```
  qsort [ y | y<-xs, y<=x ] ++
```

```
  [x] ++
```

```
  qsort [ y | y<-xs, y>x ]
```

Rezumat

Perechi și liste

Funcții

. și \$

Expresii condiționale

Expresii pentru legare locală

Evaluare leneșă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții

. și \$

Expresii condiționale

Expresii pentru legare locală

Evaluare leneșă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și \$

Expresii condiționale

Expresii pentru legare locală

Evaluare leneșă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale

Expresii pentru legare locală

Evaluare leneșă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale: if ... then ... else ..., case ... of (... -> ...), (| ... = ...)

Expresii pentru legare locală

Evaluare leneșă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale: if ... then ... else ..., case ... of (... -> ...), (| ... = ...)

Expresii pentru legare locală: let ... in ..., ... where ...

Evaluare leneșă

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale: if ... then ... else ..., case ... of (... -> ...), (| ... = ...)

Expresii pentru legare locală: let ... in ..., ... where ...

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions

Rezumat

Perechi și liste: (1,'a'), fst, snd, [1,2,3], [], :, head, tail, null, length, ++

Funcții: \x y -> corp, f x y = corp

. și \$: compunere de funcții / aplicație de funcție

Expresii condiționale: if ... then ... else ..., case ... of (... -> ...), (| ... = ...)

Expresii pentru legare locală: let ... in ..., ... where ...

Evaluare leneșă: argumentele nu se evaluează la apel, apoi se evaluează maxim o dată

List comprehensions: [expr | generatori, condiții, legări locale]
... <- ... let ... = ...