Examen PP varianta A — NOT EXAM MODE

07.06.2022

ATENȚIE: Aveți 2 ore · 1-9: 10p; 10: 30p · 100p pentru nota maximă · **Justificați** răspunsurile!

---

1. Reduceți expresia lambda $(\lambda y.\lambda x.(y\ \ y)\ \ (x\ \ x))$
   Soluție:
   $\xrightarrow{\alpha} (\lambda y.\lambda z.(y\ \ y)\ \ (x\ \ x)) \xrightarrow{\lambda y.} \lambda z.((x\ \ x)\ \ (x\ \ x))$

2. Care dintre cele două adunări se vor realiza în codul Racket de mai jos? Justificați!
   ```
   (define y 10)
   ((lambda (x) (lambda (y) (if (> x 2) (+ 1 y) (+ x y)) )) 5)
   ```
   Soluție:
   Niciuna, pentru că $\lambda y.$ nu este aplicat. Expresia întoarce rezultatul aplicării lui $\lambda x.$, care este o procedură.

3. Date fiind două liste de numere `L1` și `L2`, scrieți în Racket codul care produce o listă de perechi de forma `(x . L)`, unde `x` este un element din `L1`, iar `L` este lista elementelor din `L2` care sunt divizori ai lui `x`. E.g. pentru `L1 = (25 30 100)` și `L2 = (2 3 5)` rezultatul este `((25 . (2 5)) (30 . (2 3 5)) (100 . (2 5)))`. Nu folosiți recursivitate explicită. Explicați cum funcționează codul.
   Soluție:
   ```
   (define (divpairs L1 L2) (map
     (λ (x) (cons x (filter (λ (n) (zero? (remainder x n))) L2))) L1))
   (divpairs '(25 30 100) '(2 3 5))
   ```

4. a. Ce efect are următorul cod Racket:
   ```
   (define a (lambda (f g L) (if (null? L) '()
     (append (if (f (car L)) (list (g (car L))) '()) (a f g (cdr L))))))
   ```
   b. **Rescrieți** funcția cu funcționale, evitând recursivitatea explicită.
   Soluție:
   a. Aplică funcția g fiecărui element din L pentru care funcția f este adevărată și întoarce o listă cu rezultatele acestor aplicări.
   b. `(define (afunc f g L) (map g (filter f L)))`

5. Sintetizați tipul funcției de mai jos în Haskell. Explicați sinteza de tip pas cu pas!
   ```
   f x y = (head x, head y) : f (tail x) (tail y)
   ```
   Soluție:
   ```
   f
     f :: a → b → c
     x :: a
     y :: b
   head, tail
     x :: [d] = a
     y :: [e] = b
   cons
     c = [(d, e)]
     ⇒
   f :: [d] → [e] → [(d, e)]
   ```

6. Instanțiați clasa `Eq` pentru funcții Haskell care iau un argument numeric, astfel încât două funcții sunt "egale" dacă valoarea lor este egală pentru fiecare număr întreg între 1 și 10.
   Soluție:
   ```
   instance (Num a, Enum a, Eq b) => Eq (a -> b) where
     f == g = and (zipWith (==) (map f [1..10]) (map g [1..10]))
   ```
   Notă: `Enum a` nu era cerut.

7. Construiți în Haskell funcția `perms :: [Char] -> [[Char]]` care primește o listă de caractere și întoarce fluxul șirurilor formate din aceste caractere, începând cu șiruri de lungime 1. De exemplu: `take 45 $ perms "abc"` întoarce `["a","b","c","aa","ba","ca", ...,"bc","cc","aaa","baa","caa","aba","bba", ..., "acc","bcc","ccc","aaaa","baaa","caaa","abaa","bbaa","cbaa"]`
   Soluție:
   ```
   perms alfa = (map (:"") alfa) ++ [c:s | s <- perms alfa, c <- alfa]
   ```

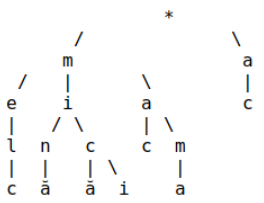
Construiesc șirurile de lungime 1, apoi celelalte șiruri sunt șiruri deja în flux la care adaug unul dintre caracterele din alfaetul dat.

8. Știind că *Cum așterne, așa doarme* și că $\neg doarme(Ion, bine)$, demonstrați **prin metoda rezoluției** că *Ion nu a așternut bine.*
Soluție:
Avem premisele: $\neg doarme(Ion, bine)$ și $\forall x.\forall y.asterne(x, y) \Rightarrow doarme(x, y)$
Concluzia: $\neg asterne(Ion, bine)$
Clauzele:
(a) $\{\neg doarme(Ion, bine)\}$
(b) $\{\neg asterne(x, y) \vee doarme(x, y)\}$
(c) $\{asterne(Ion, bine)\}$ (concluzia negată)
(b) + (c) $\{x \leftarrow Ion, y \leftarrow bine\} \rightarrow doarme(Ion, bine)$ (d)
(a) + (d) $\rightarrow$ clauza vidă

9. Construiți în Prolog un predicat `up(+L, -LUp)` care produce în `LUp` o listă cu următoarele proprietăți:
   - primul element din `LUp` este același cu primul element din lista `L`
   - următorul element din `LUp` este următorul element din `L`, mai mare decât primul.
   - următorul element din `LUp` este următorul element din `L`, mai mare decât anteriorul din `LUp`, etc.

   Exemplu: `up([5, 6, 3, 4, 8, 5, 9, 2], LUp)` leagă `LUp` la `[5, 6, 8, 9]`.
   Soluție:
```
% up(+L, -LUp)
up([H|T], [H|TUp]) :- upAux(H, T, TUp).
upAux(_, [], []).
% H face parte din secvența crescătoare:
upAux(CurrentMax, [H|T], [H|TUp]) :- H > CurrentMax, upAux(H, T, TUp).
% H nu face parte din secvența crescătoare:
upAux(CurrentMax, [H|T], TUp) :- H =< CurrentMax, upAux(CurrentMax, T, TUp).
```
   Alternativ, pot evita comparația din ultima regulă, dacă folosesc `!` după comparația din penultima regulă.

10. Un Trie (un arbore de prefixe) este un arbore care stochează cuvinte, după prefixele lor comune. De exemplu, arborele din dreapta stochează cuvintele "melc", "mină", "mică", "mici", "mac", "mama", și "ac". Fiecare nod, în afară de rădăcină. conține o literă. Rezolvați următoarele cerințe într-un limbaj la alegere dintre Racket, Haskell și Prolog:



   a1. descrieți structura de date pentru reprezentarea unui arbore de prefixe (în Haskell, definiți tipul cu `data`, în Racket și Prolog descrieți cum structurați arborele folosind construcțiile oferite de limbaj.
   a2. scrieți funcția / predicatul `emptyTrie`, care construiește un arbore de prefixe vid (care nu conține niciun cuvânt).
   b1. scrieți funcția / predicatul `canBeFollowedBy` care pentru un caracter și un nod întoarce adevărat dacă nodul are un copil cu valoarea egală cu caracterul dat. În exemplu, pentru primul nod m, funcția întoarce adevărat pentru caracterele e, și și a.
   b2. scrieți funcția / predicatul `getSubtreeFor` care pentru un caracter și un nod întoarce nodul copil al nodului dat, care are valoarea egală caracterul dat. E.g. pentru caracterul e și primul nod m din exemplu, funcția întoarce subarborele cu rădăcina în nodul e.
   c. scrieți funcția / predicatul `addWord`, care primește un cuvânt și un arbore de prefixe, și adaugă cuvântul la arbore.
   Soluție:
   Vezi fișier separat