Examen PP varianta B — 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 $(\lambda x.(\lambda x.(x\ \ x)\ \ \lambda x.x)\ \ (x\ \ x))$
   Soluție:
   $\xrightarrow{\textit{primul } \lambda x.} (\lambda x.(x\ \ x)\ \ \lambda x.x) \to (\lambda x.x\ \ \lambda x.x) \to \lambda x.x$

2. Care dintre cele două adunări se vor realiza în codul Racket de mai jos? Justificați!
   ```
   (define y 10)
   ((lambda (x) (if (> x 2) (lambda (y) (+ y 1)) (+ x y))) 5)
   ```
   Soluție:
   Niciuna, pentru că suntem pe ramura de true a if-ului, și apoi $\lambda y.$ nu se aplică. Rezultatul expresiei 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 multipli ai lui `x`. E.g. pentru `L1 = (2 3 4)` și `L2 = (4 5 8 9 10 100)` rezultatul este `((2 . (4 8 10 100)) (3 . (9)) (4 . (4 8 100)))`. Nu folosiți recursivitate explicită. Explicați cum funcționează codul.
   Soluție:
   ```
   (define (mulpairs L1 L2) (map
     (λ (x) (cons x (filter (λ (n) (zero? (remainder n x))) L2))) L1))
   (mulpairs '(2 3 4) '(4 5 8 9 10 100))
   ```

4. a. Ce efect are următorul cod Racket:
   ```
   (define b (lambda (f g L) (if (null? L) '()
     (append (if (f (g (car L))) (list (g (car L))) '()) (b f g (cdr L))))))
   ```
   b. **Rescrieți** funcția cu funcționale, evitând recursivitatea explicită.
   Soluție:
   a. Aplică funcția g pe fiecare element din L, apoi păstrează în rezultat doar acele rezultate pentru care funcția f este adevărată.
   b. `(define (bfunc f g L) (filter f (map g L)))`

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

6. Instanțiați clasa `Ord` pentru funcții Haskell care iau un argument numeric, astfel încât o funcție este "mai mică" decât alta dacă valoarea ei este mai mică decât a celeilalte funcții **în cel puțin unul** dintre numerele întregi între 1 și 10.
   Soluție:
   ```
   instance (Num a, Enum a, Ord b) => Ord (a -> b) where
     f <= g = or (zipWith (<=) (map f [1..10]) (map g [1..10]))
   ```
   Notă: `Enum a` nu era cerut.

7. Implementați în Racket fluxul în care primele 3 elemente sunt 1, 2 și 3, iar fiecare dintre următoarele elemente este produsul dintre cele trei elemente anterioare.
   Soluție:

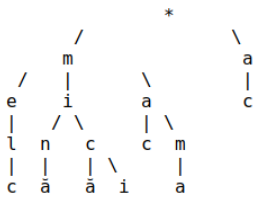
```
(define (stream-zip3 f s1 s2 s3) (stream-cons (f (stream-first s1) (stream-first s2) (stream-first
s3))
(stream-zip3 f (stream-rest s1) (stream-rest s2) (stream-rest s3)) ))
(define superProducts (stream-cons 1 (stream-cons 2 (stream-cons 3
(stream-zip3 * superProducts
(stream-rest superProducts) (stream-rest (stream-rest superProducts)) ) )))) (stream->list (stream-take
superProducts 7))
```

8. Știind că *Cine învață, nu moare de foame* și că $moareDe(Ion, foame)$, demonstrați **prin metoda rezoluției** că *Ion nu învață*.
Soluție:
Avem premisele: $moareDe(Ion, foame)$ și $\forall x.invata(x) \Rightarrow \neg moareDe(x, foame)$
Concluzia: $\neg invata(Ion)$
Clauzele:
(a) $\{moareDe(Ion, foame)\}$
(b) $\{\neg invata(x) \vee \neg moareDe(x, foame)\}$
(c) $\{invata(Ion)\}$ (concluzia negată)
(b) + (c) $\{x \leftarrow Ion\} \rightarrow \neg moareDe(Ion, foame)$ (d)
(a) + (d) $\rightarrow$ clauza vidă

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

   Exemplu: `dn([5, 6, 3, 4, 8, 5, 9, 2], LDown)` leagă `LDown` la `[5, 3, 2]`.
   Soluție:
```
% dn(+L, -LDown)
dn([H|T], [H|TDown]) :- dnAux(H, T, TDown).
dnAux(_, [], []).
% H face parte din secvența crescătoare:
dnAux(CurrentMin, [H|T], [H|TDown]) :- H < CurrentMin, dnAux(H, T, TDown).
% H nu face parte din secvența crescătoare:
dnAux(CurrentMin, [H|T], TDown) :- H >= CurrentMin, dnAux(CurrentMin, T, TDown).
```
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 `countWords`, care numără cuvintele stocate într-un arbore de prefixe. În exemplu sunt 7 cuvinte.
b2. scrieți funcția / predicatul `printWords`, care întoarce o listă cu cuvintele stocate într-un arbore de prefixe. Considerăm un cuvânt ca o cale de la rădăcină la o frunză. Pentru Racket și Prolog, folosiți o reprezentare a cuvintelor mai ușor de implementat, nu trebuie neapărat să fie șiruri de caractere.
c. scrieți funcția / predicatul `predict`, care pentru un prefix și un arbore de prefixe, întoarce o listă cu toate cuvintele care conțin prefixul dat.
Soluție:
Vezi fișier separat