

1. Reduceți expresia lambda până la forma normală, **ilustrând** pașii reducerii:
($\lambda x.(x \lambda x.y) \lambda x.x$)
2. Cum se evaluează codul Racket de mai jos? **Explicați** la ce se leagă fiecare variabilă și care este rezultatul final.
 1. (define x 10)
 2. (define y 20)
 3. (define (ev a) (if (promise? a) (force a) a))
 4. (let* (
 5. (x (add1 (ev y)))
 6. (y (delay (add1 x)))
 7. (z (+ x (ev y))))
 8. (+ x (ev y) z))
3. Implementați în Racket sau în Haskell funcția `lookup` care primește o listă de perechi cheie - valoare și o listă de chei și întoarce o listă cu valorile din prima listă ale căror chei se regăsesc în a doua listă. **Nu utilizați recursivitate explicită și utilizați cel puțin o funcțională.** De exemplu, în Racket, apelul (`lookup '(a . 1) (b . 2) (c . 3) (d . 4) (e . 5) (f . 6)`) '(b f c d) trebuie să întoarcă lista '(2 3 4 6)
4. Construiți în Racket sau în Haskell fluxul de perechi între numere naturale și lista lor de divizori. Fluxul începe (în Haskell) cu [(1, [1]), (2, [1, 2]), (3, [1, 3]), (4, [1, 2, 4]), (5, [1, 5]), (6, [1, 2, 3, 6]) ...].
5. Sintetizați, **ilustrând** procesul de sinteză, tipul funcției Haskell `f = foldr (.) (+1)`
6. **Explicați** care este valoarea expresiei de mai jos și câte adunări se realizează pentru a o calcula:
`let fib = 0 : 1 : (zipWith (+) fib $ tail fib) in head $ filter (< 20) $ filter (> 10) fib`
7. Se dă tipul funcțiilor unare care întorc o valoare de același tip cu argumentul:
`data UnaryF a = F (a -> a)` Instanțiați clasa `Num` pentru tipul de mai sus, implementând în cadrul instanței operatorul `*`, care aplicat unor funcții `f` și `g` are ca rezultat construirea unei noi funcții unare care pentru orice argument întoarce suma valorilor lui `f` și lui `g` pentru acel argument.
8. Știind că “Câinele nu mușcă mâna care îl hrănește”, că Grivei este câine, și că Grivei mușcă mâna lui George, demonstrați **prin rezoluție și folosind reducerea la absurd** că mâna lui George nu îl hrănește pe Grivei.
9. Folosiți **metapredicate** pentru a implementa predicatul `p(L, R)`, care pentru `L` o listă de liste numerice, filtrează în `R` acele liste din `L` care încep cu elementul lor maxim. De exemplu, `p([[1,2,3], [5,3,2], [3,2,1,4], [6,4,3,2,1], [2,1], [1,2]], R)` leagă `R` la lista `[[5,3,2], [6,4,3,2,1], [2,1]]`.
10. PROBLEMA (Poate fi implementată în orice limbaj studiat la PP.) Se urmărește implementarea unui *multi-set*, o mulțime în care valorile pot apărea de mai multe ori. De exemplu, putem avea un multiset M în care valoarea a apare de 2 ori, valoarea b o dată, și valoarea c de 5 ori, în total cardinalul mulțimii fiind de 8. În implementare, **evitați recursivitatea explicită**.
 - (a) Descrieți reprezentarea *multi-set*-ului. Pentru Haskell, dați definiția tipului de date polimorfic. Reprezentați mulțimea *eficient*, **mai concis** decât o listă cu toate aparițiile elementelor (pentru mulțimea M , mai concis decât o listă cu 8 elemente). Definiți funcția/predicatul `find`, care extrage numărul de apariții ale unui element în mulțime (în Haskell întoarce o valoare de tipul `Maybe`), iar în cazul în care elementul nu aparține mulțimii, întoarceți `#f`, `Nothing`, respectiv `false`.
 - (b) Definiți funcția/predicatul `cardinal`, care întoarce cardinalul mulțimii. Definiți funcția/predicatul `add`, pentru adăugarea unui element la set. De exemplu, dacă în setul M se adaugă valoarea c , atunci în rezultat valoarea c va apărea de 6 ori.
 - (c) Definiți funcția/predicatul `intersect`, care realizează intersecția a două *multi-seturi*, numărul de apariții ale elementelor comune în rezultat fiind minimul dintre numărul de apariții între cele două mulțimi inițiale. De exemplu,, pentru setul $M1$ conținând b de 2 ori, a o dată, și d de 3 ori intersecția $M \cap M1$ are ca rezultat o mulțime care conține a o dată și b o dată.