Nume: ______________________________ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

— NOT EXAM MODE

Examen PP CC var B-D 15.06.2023

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

---

1. Reduceți expresia lambda până la forma normală, **ilustrând** pașii reducerii:
   $(\lambda x.(x\ \ \lambda x.y)\ \ \lambda x.x)$
   Soluție:
   $(\lambda x.(x\ \ \lambda x.y)\ \ \lambda x.x) \rightarrow (\lambda x.x\ \ \lambda x.y) \rightarrow \lambda x.y$
   [4p / reducere corectă + 2p ajungere / oprire la rezultatul corect]

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))
   ```
   Soluție:
   x din let: y (din define) + 1 = 21
   y din let: promisiune pentru calculul lui x din let + 1
   z din let: x (din let) + y (din let, se forțează evaluarea la 22) = 43
   corp let: 21 + 22 + 43 = 86
   [2p fiecare valoare corectă pentru x, y, z din let; 2p valoarea finală; 2p explicații]

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)`
   Soluție:
   ```
   (map cdr (filter (λ (p) (member (car p) L2)) L1))
   ```
   [4p căutarea perechii după cheie / căutarea cheii pentru o pereche; 4p selectarea valorilor; 2p construcție corectă]

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])...]`.
   Soluție:
   ```
   naturals = [1..]
   mults = zip naturals $ map (divs naturals) naturals
      where divs xs n = [x | x <- take n xs, n `mod` x == 0]
   ```
   [2p construcție flux, 3p construcție perechi, 5p lista de divizori]

5. Sintetizați, **ilustrând** procesul de sinteză, tipul funcției Haskell `f = foldr (.) (+1)`
   Soluție:
   foldr :: Foldable t ⇒ (a → b → b) → b → t a → b
   (.) :: (d → e) → (c → d) → c → e

   foldr primește (.) ca prim argument, deci
   a → b → b ≡ (d → e) → (c → d) → c → e
   b ≡ c → d ≡ c → e deci e = d
   a ≡ d → e adică a = d → d

   (+1) :: Integer → Integer
   foldr primește (+1) ca al doilea argument, deci
   b ≡ Integer → Integer

dar b = c → d
deci c = d = e = Integer

f este rezultatul aplicării lui foldl pe 2 argumente, deci f :: t a → b
unde a = Integer → Integer
iar b = Integer → Integer
putem considera t a ca fiind [a]

f :: [Integer → Integer] → Integer → Integer

PS: Haskell va da eroare la sinteza de tip, puteți adăuga definiția `g = f . tail` pentru a concretiza instanța lui Foldable. PPS: contextul `Foldable t` nu era obligatoriu, puteți presupune că `fold` funcționează direct pe liste
[1p tip fold, 1p tip (.), 1p tip (+), 1p tip (+1), 1p tip fold + funcție, 1p per restul de unificări corecte]

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`
Soluție:
Se fac adunările:
$0 + 1 = 1$
$1 + 1 = 2$
$1 + 2 = 3$
$2 + 3 = 5$
$3 + 5 = 8$
$5 + 8 = 13$
În acest moment avem prima valoarea mai mare decât 10 (și mai mică decât 20) și `head` poate întoarce o valoare. Nu mai este nevoie de alte calcule. Rezultatul este 13.
[5p număr corect de adunări și rezultat final, 5p explicații]

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.
Soluție:
```
instance Num a => Num (UnaryF a) where
   F f * F g = F $ \x -> f x + g x
```
[2p antet, 2p Num a, 3p constructori de date utilizați corect, 2p definiție funcție rezultat, 1p corp funcție rezultat]

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.
Soluție:
Propoziții:
$\forall x, y. câine(x) \wedge hrănește(y, x) \Rightarrow \neg mușcă(x, y)$ cu FNC $\neg câine(x) \vee \neg hrănește(y, x) \vee \neg mușcă(x, y)$
$câine(Grivei)$ (2)
$mușcă(Grivei, Mâna\_George)$ (3)
Adăugăm concluzia negată: $hrănește(Mâna\_George, Grivei)$(4)
(1) + (4) cu $\{x \leftarrow Grivei, y \leftarrow Mâna\_George\} \rightarrow \neg câine(Grivei) \vee \neg mușcă(Grivei, y)$
+ (3) cu $\rightarrow \neg câine(Grivei)$
+ (2) → clauza vidă
[4p traducerea în FOL, 2p FNC, 4p rezoluția]

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]]`.
Soluție:
```
p(L, R) :- findall(X, ( member(X, L), X = [H | _], forall(member(M, X), M =< H) ), R).
```
[6p `findall` (2p forma, 2p member, 2p extragere H), 4p `forall` (sau altă soluție echivalentă, e.g. `sort`

sau `max_list`)]

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 ilistă 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.

    Soluție:
    vezi fișier separat