Examen PP CB A — NOT EXAM

MODE15.06.26 Grupa: ____

Nume: _____

1	2	3	4	5	6	7	8	9	a	b	c

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

---

1. Aplicați funcția $\lambda f.(\lambda x.(f\ \ (x\ \ x))\ \ \lambda x.(f\ \ (x\ \ x)))$ pe $\lambda x.x$. Ilustrați 3 pași de reducere stânga-dreapta.
   Soluție:
   $(\lambda f.(\lambda x.(f\ \ (x\ \ x))\ \ \lambda x.(f\ \ (x\ \ x)))\ \ \lambda x.x) \rightarrow$
   $(\lambda x.(\lambda x.x\ \ (x\ \ x))\ \ \lambda x.(\lambda x.x\ \ (x\ \ x))) \rightarrow$
   $(\lambda x.x\ \ (\lambda x.(\lambda x.x\ \ (x\ \ x))\ \ \lambda x.(\lambda x.x\ \ (x\ \ x)))) \rightarrow$
   $(\lambda x.(\lambda x.x\ \ (x\ \ x))\ \ \lambda x.(\lambda x.x\ \ (x\ \ x)))$

2. Se dau tipurile `CList<T>` `NestList <T>` cu

   constructorii de bază:
   `atom : T → NestList <T>`
   `list : CList<T> → NestList <T>`
   `null : CList<T>`
   `cons : NestList<T> × CList<T> → CList<T>`

   și operatorii:
   `join : NestList<T> × NestList<T> → NestList<T>`
   `size : NestList<T> → Int`

   Scrieți axiomele pentru `join` și `size`, știind că `size` întoarce numărul de atomi, iar `join` construiește un nivel nou de `CList` doar atunci când ambii operanzi sunt atomi.
   Soluție:

```
size(atom(x)) = 1
size(list(null)) = 0
size(list(cons(nl, cl))) = size(nl) + size(list(cl))

join(atom(x), atom(x)) = list(cons(atom(x), cons(atom(y), null)))
join(atom(x), list(cl)) = list(cons(atom(x), cl))
join(list(cl1), nl2) = list(app(cl1, nl2))
app(null, atom(x)) = cons(atom(x), null)
app(null, list(cl)) = cl
app(cons(nl1, cl), nl2) = cons(nl1, app(cl, nl2))
```

   cu `app : CList<T> × NestList<T> → CList<T>`

3. Date fiind funcțiile `f` și `g` deja definite și următorul cod care considerăm că se execută fără erori, de câte ori sunt aplicate fiecare dintre cele două funcții, și la ce linii din cod se fac evaluările?

```
1.  (define a (λ (x)
2.     (let* [   (f (f x))   (g (delay (f x)))   ]
3.     g)   ))
4.  (a g)
```

   Soluție:
   `f` se apelează o dată la linia 2; `g` nu se apelează niciodată

4. Implementați în Racket **fără recursivitate explicită** funcția `picks` care aplicată pe o listă de funcții unare `xs` și o listă de numere `L` întoarce o listă în care avem pentru fiecare element `e` din `L` minimul dintre rezultatele aplicării funcțiilor din `xs` pe `e`.
   Soluție:

```
(define (pick xs L) (map (lambda (e) (apply min (map (lambda (f) (f e)) xs))) L))
```

5. Explicați ce face funcția Racket:

```
(compose length ((curry filter) not) ((curry map) ((curry member) 5)))
```

   Soluție:
   Funcția rezultată din compunere se aplică pe o listă de liste (pentru că avem `map` cu `member`) și calculează câte liste (`length`) din lista de liste (filtrate cu `filter`) nu conțin (`member`) numărul 5.

6. Construiți un flux Haskell cu rezultatele aplicărilor succesive ale funcției `f` pe un număr `x` (`x, f x, f (f x)...`) și folosiți-l pentru a afla de câte aplicări este nevoie pentru a ajunge la un rezultat egal cu rezultatul precedent.
   Soluție:

```
flux f x = length $ takeWhile (/= 0) $ iterate f x
```

(sau length + 1)

Pentru următoarele două exerciții se dă tipul `data Tree a = Nil | Node a [Tree a]`

7. Dată fiind clasa Haskell `class TreeLike a where toTree :: a -> Tree a`, instanțiați clasa pentru numere întregi, astfel încât arborele corespunzător unui număr va avea numărul în rădăcină și doi fii, cel din stânga rezultatul împărțirii întregi la doi și cel din dreapta diferența care rămâne (e.g. 5 va avea copiii 2 și 3).
   Soluție:
```
instance TreeLike Integer where
  toTree x
    | x <= 1 = Leaf x
    | True = Node x [toTree hx, toTree $ x - hx]
    where hx = div x 2
```

8. Date fiind definițiile și definițiile de la problemele 7 și 10, sintetizați, **ilustrând** procesul de sinteză, tipul funcției `map (toTree . List)`
   Soluție:
   `toTree :: TreeLike b ⇒ b → Tree b`
   `List :: [NestedList a] → NestedList a`
   deci
   `toTree . List :: [NestedList a] → Tree (NestedList a)` cu `TreeLike (NestedList a)`
   `map :: (f → g) → [f] → [g]`
   deci
   `f = [NestedList a]`
   `g = Tree (NestedList a)` astfel
   `map (toTree . List) :: TreeLike (NestedList a) ⇒ [[NestedList a]] → [Tree (NestedList a)]`

9. Folosind raționament ecuațional, simplificați pentru a folosi un singur `map` și un singur `filter` expresia
   `map a . filter b . map c.`
   Soluție:
   `map a . filter b . map c →`
   `map a . map c . filter (b.c) →`
   `map (a.c) . filter (b.c)`

10. Dat fiind tipul de date *listă imbricată* `data NestedList a = Atom a | List [NestedList a]`, ne dorim să construim în Haskell o formă arborescentă care să ne ajute la diverse prelucrări. O prelucrare va primi lista și va întoarce un *arbore* care conține rezultatele parțiale ale prelucrărilor pentru fiecare listă individuală din lista imbricată. E.g. pentru lista `nl = List [Atom 1, Atom 2, List [Atom 3, Atom 4, Atom 5], List [Atom 6, Atom 7]]` vom dori să facem suma elementelor obținând rezultatele parțiale 12 (pentru primul `List`), 13 (pentru al doilea), și apoi rezultatul complet 28, *reținând* în arbore aceste rezultate.
    (a) definiți tipul `ResultTree a` care ne permite ca în fiecare frunză să reținem rezultatul unei prelucrări pentru un element atomic din lista imbricată iar în fiecare nod să reținem rezultatul unei prelucrări corespunzător unei valori de tip `List` din lista imbricată. Definiți funcția
    `compute :: ([a] -> a) -> (t -> a) -> NestedList t -> ResultTree a`
    care construiește arborele de rezultate.
    (b) definiți funcția `params` care primește o listă imbricată și folosește `compute` pentru a produce un triplu dintre minimul și maximul peste elementele listei și adâncimea listei. `params nl = (1, 7, 2)`
    (c) definiți funcția `params2` care face același lucru cu `params`, dar parcurge lista *o singură dată*.
    Soluție:
    vezi `problema.hs`