Precizări:

- Primele 9 subiecte au fiecare câte 10p. Cele 3 cerințe ale problemei au fiecare câte 10p. **Punctajul NU se acordă în absența justificării răspunsului!**
- Este suficientă rezolvarea a **10** itemi dintre cei 12 pentru nota maximă.
- Punctajul suplimentar reprezintă **bonus**, ce poate compensa punctajul de pe parcurs.

1. Fie λ-expresia $E \equiv \lambda x.(x\ \lambda x.x)$. Evaluați **pas cu pas** aplicația $((E\ E)\ E)$.

   Soluție.

   Aparițiile libere în raport cu corpul funcției aplicate, care urmează a fi substituite, sunt subliniate.

   $$\begin{aligned}
   (E\ E) &\to (\lambda x.(\underline{x}\ \lambda x.x)\ \lambda x.(x\ \lambda x.x)) \\
   &\to (\lambda x.(\underline{x}\ \lambda x.x)\ \lambda x.x) && \checkmark \\
   &\to (\lambda x.\underline{x}\ \lambda x.x) && \checkmark \\
   &\to \lambda x.x && \checkmark \\
   ((E\ E)\ E) &\to (\lambda x.\underline{x}\ E) \\
   &\to E \\
   &\to \lambda x.(x\ \lambda x.x) && \checkmark \qquad \square
   \end{aligned}$$

   Barem.

   2,5p $\times$ 4 pași bifați

2. Definiți în Racket funcția `count`, care primește o listă și numără câte elemente constituie mijlocul unui palindrom de lungime $2n + 1$, unde n este valoarea mijlocului. Exemple, unde mijlocurile sunt subliniate:

   - `(count '(1))` $\to$ `0`
   - `(count '(2 1 2))` $\to$ `1`
   - `(count '(3 7 2 7 3 7 2 7))` $\to$ `2`
     (adică `'(3 7 2 7 3)` și `'(7 2 7 3 7 2 7)`).

   (a) Implementați funcția `count` utilizând **recursivitate explicită**.

   (b) Menționați și justificați tipul de recursivitate folosit.

   Soluție.

   (a) Implementare:

```
1  (define (count L [left '()] [n 0])
2    (match L
3      ['() n]
4      [(cons x xs) (count xs
5                          (cons x left)
6                          (+ n
7                             (if (and (<= x (length xs))
8                                      (<= x (length left))
```

```
9                                       (equal? (take xs x)
10                                              (take left x)))
11                                  1
12                                  0)))]))
```

(b) Implementarea de mai sus folosește recursivitate pe coadă, întrucât nu se mai desfășoară alte operații după evaluarea apelului recursiv. □

Barem.

(a) 8p
- 1p lista vidă
- 2p calcularea porțiunii din stânga elementului curent
- 2p asigurarea lungimii suficiente a porțiunilor din stânga și din dreapta elementului curent
- 2p compararea porțiunilor din stânga și din dreapta elementului curent
- 1p contorizarea

(b) 2p
- 1p tip de recursivitate
- 1p justificare

3. Utilizând **exclusiv funcționale**, definiți în Racket funcția `combine`, care primește o listă de funcții și o listă de parametri ai acestora (nevide, cu aceeași lungime), aplică parțial fiecare funcție pe parametrul de la aceeași poziție, și compune funcțiile rezultate. De exemplu, valoarea expresiei `(combine (list map filter) (list add1 odd?))` este o funcție a cărei aplicație asupra listei `L` este echivalentă cu `(map add1 (filter odd? L))`.

Soluție.

```
1 (define combine
2   (compose (curry apply compose)
3            (curry map curry)))
```
□

Barem.

- 3p parcurgerea simultană a listelor de funcții și de parametri
- 2p aplicarea parțială a fiecărei funcții pe parametrul corespunzător
- 3p compunerea funcțiilor rezultate
- 2p ordonarea corectă a celor două etape

4. Definiți în Racket funcția `picks`, care primește un flux infinit și construiește un flux infinit de fluxuri infinite, în care fluxul interior de pe poziția $i \geq 0$ se obține pornind de la fluxul inițial și trăgând în față elementul de pe poziția i (toate fluxurile interioare folosesc întotdeauna ca punct de plecare fluxul inițial). De exemplu, `(picks naturals)` → `(stream (stream 0 1 2 3 ...) (stream 1 0 2 3 ...) (stream 2 0 1 3 ...) (stream 3 0 1 2 ...) ...)`.

Soluție.

```
1 (define (picks xs)
2   (stream-cons
3    xs
4    (stream-map (lambda (ys)
5                  (stream-cons (stream-first ys)
6                               (stream-cons (stream-first xs)
7                                            (stream-rest ys))))
8                (picks (stream-rest xs)))))
```
□

Barem.

- 3p introducerea fluxului inițial ca prim element al fluxului rezultat
- 3p construirea recursivă pentru restul fluxului inițial
- 4p adăugarea primului element al fluxului inițial pe a doua poziție în fiecare flux interior obținut recursiv

5. Sintetizați tipul expresiei Haskell `(concat . concat)`. În interpretor, comanda `:t concat` afișează `Foldable t => t [a] -> [a]`.

Soluție.

Distingem între cele două apariții ale lui `concat`:

```
1  concat_1 :: Foldable t => t [a] -> [a]
2  concat_2 :: Foldable u => u [b] -> [b]
3  (.)      :: (d -> e) -> (c -> d) -> c -> e
4
5  d -> e  =  Foldable t => t [a] -> [a]
6  d = Foldable t => t [a]
7  e = [a]
8
9  c -> d  =  Foldable u => u [b] -> [b]
10 c = Foldable u => u [b]
11 d = [b]
12
13 [b] = Foldable t => t [a]
14 t = []  -- constructorul de tip lista, nu lista vida
15 b = [a]
16
17 c -> e = Foldable u => u [[a]] -> [a]
```
□

Barem.

- 1p tip prima apariție `concat` (linia 1)
- 1p tip a doua apariție `concat` (linia 2)
- 1p tip `(.)` (linia 3)
- 2p unificare tip parametru 1 `(.)` cu tip prima apariție `concat` (liniile 5–7)
- 2p unificare tip parametru 2 `(.)` cu tip a doua apariție `concat` (liniile 9–11)
- 2p unificare forme aferente lui `d` (liniile 13–15, vizând liniile 6 și 11)

- 1p tipul final (linia 17)

6. Fie în Haskell clasa `Zippable`, care surprinde structuri care pot fi combinate element cu element. Cum ar putea fi instanțiată clasa pentru constructorul `(c -> _)` (adică constructorul de tip funcție, aplicat parțial pe tipul parametrului, așteptând în continuare tipul rezultatului)? Evidențiați tipul particularizat al funcției `genericZip`.

```
1 class Zippable z where
2     genericZip :: z a -> z b -> z (a, b)
```

Soluție.

```
1 instance Zippable ((->) c) where
2     genericZip :: (c -> a) -> (c -> b) -> c -> (a, b)
3     genericZip f g c = (f c, g c)
```
□

Barem.

- 0,5p antetul clasei, neaplicând complet constructorul de tip (se acceptă și varianta `(c ->)`, deși nu este validă în Haskell)
- 1,5p tip particularizat `genericZip`
- 8p implementare corectă
  - 3 × 1p cei trei parametri
  - 2p aplicația primei funcții
  - 2p aplicația celei de-a doua funcții
  - 1p producerea perechii

7. Fie tipul de date abstract *arbore binar cu chei de tipul* K, $BinTree\langle K\rangle$, caracterizat de constructorii $empty : BinTree\langle K\rangle$ și $node : K \times BinTree\langle K\rangle \times BinTree\langle K\rangle \to BinTree\langle K\rangle$, și de operatorii $size : BinTree\langle K\rangle \to \mathbb{N}$ și $height : BinTree\langle K\rangle \to \mathbb{N}$, care calculează dimensiunea, respectiv înălțimea unui arbore binar.

   (a) Scrieți axiomele pentru operatorii *size* și *height*. Considerați că înălțimea arborelui vid este 0.

   (b) Demonstrați prin inducție structurală că, pentru orice arbore binar t, $height(t) \leq size(t)$. Puteți manipula obișnuit numerele naturale.

Soluție.

(a) Axiome *size*:

$$size(empty) = 0$$
$$size(node(k, l, r)) = 1 + size(l) + size(r)$$

Axiome *height*:

$$height(empty) = 0$$
$$height(node(k, l, r)) = 1 + \max(height(l), height(r))$$

(b) Demonstrație prin inducție structurală:

- Cazul de bază: $t = empty$. Cum $height(empty) = 0$ și $size(empty) = 0$, avem că $height(empty) \leq size(empty)$.
- Pasul inductiv: $t = node(k, l, r)$, cu ipoteza inductivă $height(l) \leq size(l)$ și $height(r) \leq size(r)$. Atunci:

$$\begin{aligned} height(node(k,l,r)) &= 1 + \max(height(l), height(r)) \\ &\leq 1 + height(l) + height(r) && \text{(deoarece } \max(a,b) \leq a+b) \\ &\leq 1 + size(l) + size(r) && \text{(ipoteza inductivă)} \\ &= size(node(k,l,r)) && \square \end{aligned}$$

Barem.

(a) 2p
- 1p axiome *size*
- 1p axiome *height*

(b) 8p
- 2p cazul de bază
  - 1p *empty*
  - 1p concluzia
- 6p pasul inductiv
  - 1p $node(k, l, r)$
  - 1p ipoteza inductivă pt **ambii** subarbori
  - 4 × 1p cei patru pași de calcul

8. Utilizând proprietatea de **fuziune** a funcționalei `foldr`, rescrieți următoarea funcție din Haskell ca o aplicație parțială a lui `foldr`: `concat . concat = foldr g b`.

Soluție.

Pentru a putea aplica proprietatea de fuziune, trebuie mai întâi să rescriem funcția din dreapta utilizând `foldr`:

```
1   concat . concat
2 = concat . foldr (++) []
```

Instanțiind variabilele din proprietate obținem:

```
1 h = concat
2 f = (++)
3 a = []
```

Prima condiție a proprietății:

```
1 b = h a = concat [] = []
```

A doua condiție a proprietății:

```
1 h (f x y) = g x (h y)
2 concat (x ++ y) = g x (concat y)
3 concat x ++ concat y = g x (concat y)
```

Generalizând `(concat y)` la `z`, obținem:

```
g x z = concat x ++ z
```

Forma finală:

```
concat . concat
= foldr (\x z -> concat x ++ z) []
= foldr ((++) . concat) []
```
□

Barem.

- 1p rescriere `concat` cu `foldr`
- 3 × 0,5p identificare variabile `h`, `f`, `a`
- 1,5p prima condiție
  - 1p aplicare condiție
  - 0,5p calcul `b`
- 5p a doua condiție
  - 2p aplicare condiție
  - 2p pasul 2–3
  - 1p definiție `g`
- 1p forma finală (cu lambda sau fără pentru `g`)

9. Ținând cont de strategia de evaluare din fiecare limbaj, precizați și explicați numărul de vizitări de elemente ale listelor în timpul evaluării fiecăreia dintre următoarele expresii:

   (a) Racket:

```
(car (append (append '(1 2)
                     '(3 4))
             '(5 6)))
```

   (b) Haskell:

```
head (([1,2] ++ [3,4]) ++ [5,6])
```

Soluție.

(a) În Racket, evaluarea este *eager*. Ea decurge astfel:
- `(append '(1 2) '(3 4))` vizitează cele două elemente ale primei liste.
- `(append '(1 2 3 4) '(5 6))` vizitează cele patru elemente ale primei liste.
- `(car '(1 2 3 4 5 6))` vizitează primul element al listei.
- În total, există $2 + 4 + 1 = 7$ vizitări.

(b) În Haskell, evaluarea este lenesă. Ea decurge astfel:
- `([1,2] ++ [3,4])` vizitează elementul `1` pentru a produce `1:([2] ++ [3,4])`.
- `1:([2] ++ [3,4]) ++ [5,6]` vizitează iarăși elementul `1` pentru a produce `1:(([2] ++ [3,4]) ++ [5,6])`.
- `head (1:(([2] ++ [3,4]) ++ [5,6]))` vizitează iarăși elementul `1` pentru a produce `1`.
- În total, există $1 + 1 + 1 = 3$ vizitări.

□

Barem.

(a) 5p

- 2p primele două vizitări
- 2p următoarele patru vizitări
- 1p ultima vizitare

(b) 5p

- 2p prima vizitare
- 2p a doua vizitare
- 1p a treia vizitare

10. PROBLEMA. În Haskell, o reprezentare alternativă a unei liste cu tipul `[a]` este constituită de o funcție cu tipul `Int -> a`, care întoarce pentru parametrul $i \geq 0$ elementul de pe poziția i din listă. Mai precis, utilizăm reprezentarea de mai jos a listelor finite, surprinsă de tipul `FList a` (*function-based list*), unde al doilea câmp reprezintă lungimea listei:

```
data FList a = FList (Int -> a) Int
```

În cerințele de mai jos, cu o excepție la punctul (c), este **INTERZISĂ utilizarea listelor standard**.

(a) Dacă utilizăm liste standard pentru a reprezenta lista primelor 10 numere naturale, lista primelor 10 numere pare, și lista de lungime 10 cu repetiții alternante ale numerelor `2` și `7` (`2, 7, 2, 7, ...`), dezavantajul este că accesul aleator se realizează în timp liniar. Definiți cele trei liste de mai sus utilizând noua reprezentare, astfel încât accesul aleator să se realizeze în timp **constant**.

(b) Implementați eficient funcțiile `reverse :: FList a -> FList a` și `append :: FList a -> FList a -> FList a`, cu semnificațiile obișnuite.

(c) Instanțiați clasele `Show` și `Functor` pentru tipul `FList a`, astfel încât `show` să evidențieze elementele listei, iar `fmap` să reflecte funcția `map` pentru listele standard. **Puteți utiliza liste standard doar în implementarea funcției `show`**.

Soluție.

```
import Prelude hiding (reverse)

data FList a = FList (Int -> a) Int

naturals :: FList Int
naturals = FList id 10

evens :: FList Int
evens = FList (2 *) 10

alternating :: FList Int
alternating = FList (\i -> if even i then 2 else 7) 10

reverse :: FList a -> FList a
reverse (FList f n) = FList (f . (n-1 -)) n
```

```
16
17 append :: FList a -> FList a -> FList a
18 append (FList f1 n1) (FList f2 n2) =
19     FList (\i -> if i < n1 then f1 i else f2 (i - n1)) (n1 + n2)
20
21 instance Show a => Show (FList a) where
22     show :: Show a => FList a -> String
23     show (FList f n) = show [f i | i <- [0 .. n-1]]
24
25 instance Functor FList where
26     fmap :: (a -> b) -> FList a -> FList b
27     fmap g (FList f n) = FList (g . f) n                                    □
```

Barem.

(a) 10p

- 2p `naturals`
- 3p `evens`
- 5p `alternating`

(b) 10p

- 4p `reverse`
  - 2p scăderea din `n-1`
  - 1p combinarea cu `f`
  - 1p păstrarea lungimii
- 6p `append`
  - 2p depistarea interiorului primei sau celei de-a doua liste
  - 1p tratarea interiorului primei liste
  - 2p tratarea interiorului celei de-a doua liste
  - 1p noua lungime

(c) 10p

- 5p `Show`
  - 1p instanțiere cu `FList a`, nu cu `FList`
  - 1p constrângere `Show a`
  - 1p enumerarea indicilor de la `0` la `n-1`
  - 1p aplicarea funcției pe fiecare indice
  - 1p obținerea șirului final
- 5p `Functor`
  - 1p instanțiere cu `FList`, nu cu `FList a`
  - 3p compunere funcții
  - 1p păstrarea lungimii