Examen Paradigme de Programare
Seria CC — 19.06.2026 — **Varianta A**
Timp de lucru: 2 ore

1	2	3	4	5	6	7	8	9	10a	10b	10c	Total

Nume și grupă: _____

---

1. Pentru TDA-ul $\text{BT}\langle K\rangle$ (arbore binar cu chei de tip K) cu constructorii de bază:

$\text{Nil} :\to \text{BT}\langle K\rangle$
$\text{Nod} : \text{BT}\langle K\rangle \times K \times \text{BT}\langle K\rangle \to \text{BT}\langle K\rangle$

și operatorii definiți prin axiomele:

$\text{mirror}(\text{Nil}) = \text{Nil}$
$\text{mirror}(\text{Nod}(S, v, D)) = \text{Nod}(\text{mirror}(D), v, \text{mirror}(S))$
$\text{fmap}(f, \text{Nil}) = \text{Nil}$
$\text{fmap}(f, \text{Nod}(S, v, D)) = \text{Nod}(\text{fmap}(f, S), f(v), \text{fmap}(f, D))$

demonstrați prin inducție structurală proprietatea:

$$\text{mirror}(\text{fmap}(f, T)) = \text{fmap}(f, \text{mirror}(T)), \quad \forall T \in \text{BT}\langle K\rangle$$

2. Fie expresia în calculul lambda: $E = y$. Dați 3 λ-expresii diferite (care nu diferă doar prin numele variabilelor) care se reduc într-un singur pas de beta-reducere la expresia E.

3. Care dintre cele 3 variante reprezintă output-ul real generat de utilitarul trace în urma rulării codului Racket de mai jos? Indicați, justificând pe scurt, cel puțin o eroare în celelalte variante.

```
(require (lib "trace.ss"))
(define (find x L)
  (cond
    ((null? L) #f)
    ((list? (car L)) (or (find x (car L)) (find x (cdr L))))
    ((equal? x (car L)) #t)
    (else (find x (cdr L)))))
(trace find)
(find 'a '(b (b (a c) d)))
```

Varianta a)

```
>(find 'a '(b (b (a c) d)))
>(find 'a '((b (a c) d)))
> (find 'a '(b (a c) d))
> (find 'a '((a c) d))
> >(find 'a '(a c))
< <#t
< #t
<#t
#t
```

Varianta b)

```
>(find 'a '(b (b (a c) d)))
>(find 'a '((b (a c) d)))
> (find 'a '(b (a c) d))
> (find 'a '((a c) d))
> >(find 'a '(a c))
< <#t
> >(find 'a '(d))
> >(find 'a '())
< <#f
< #t
> (find 'a '())
< #f
<#t
#t
```

Varianta c)

```
>(find 'a '(b (b (a c) d)))
> (find 'a '((b (a c) d)))
> >(find 'a '(b (a c) d))
> > (find 'a '((a c) d))
> > >(find 'a '(a c))
< < <#t
< < #t
< <#t
< #t
<#t
#t
```

4. Se dă un număr natural n. În Racket, folosind doar funcționale (fără recursivitate explicită) și funcția `(range n)`, generați matricea identitate de dimensiune $n \times n$ (reprezentată ca o listă de liste).
Notă: `(range n)` generează o listă cu numerele naturale de la 0 la $n-1$.
Exemplu: Pentru $n = 3$, rezultatul este `'((1 0 0) (0 1 0) (0 0 1))`.

5. Ce se întâmplă la execuția următorului cod Racket? Justificați.

```
(define x 1)
(define (plus-x y) (+ x y))
(letrec ((y (delay x)) (x 2)) (plus-x (force y)))
```

6. În Haskell, definiți un flux de fluxuri în care al n-lea flux repetă la infinit o secvență de n de 0 urmată de n de 1.
Notă: Rezultatul este un flux infinit de fluxuri infinite: `[[0,1,0,1,0,1, ...] ,[0,0,1,1,0,0, ...], [0,0,0,1,1,1, ...], ...]`.

7. Sintetizați tipul expresiei Haskell `e = uncurry . flip`.
Notă: `uncurry` transformă o funcție binară curry într-o funcție echivalentă uncurry, iar `flip` inversează ordinea celor două argumente ale unei funcții binare.

8. În Haskell, definiți tipul de date listă care alternează valori de tip `a` și `b` (începând cu tipul `a`, dacă lista este nevidă). Folosiți apoi acest tip pentru a reprezenta lista `[1, 'a', 2, 'b', 3]`.

9. Utilizând proprietatea de fuziune a funcționalei `foldr`, rescrieți funcția Haskell `f = (sum . map length)` ca o aplicație parțială a lui `foldr`. (Practic, determinați `g` și `b` pentru care `sum . map length = foldr g b`).

10. Se dă tipul `MList a`, care reprezintă o listă mixtă (m-listă), capabilă să stocheze atât elemente individuale, cât și liste Haskell standard:

```
data MList a = Null | A a (MList a) | L [a] (MList a) deriving Show
```

Rezolvați următoarele cerințe, precizând și **signaturile tuturor funcțiilor implementate**:

a) Implementați funcția `mFlatten` care transformă o structură `MList a` într-o structură echivalentă în care toate listele interne (introduse de constructorul `L`) sunt desfăcute în noduri individuale de tip `A`, păstrând ordinea.
Exemplu: `mFlatten (A 3 (L [4, 5, 6] (L [9, 8, 7] (A 1 Null))))` → `A 3 (A 4 (A 5 (A 6 (A 9 (A 8 (A 7 (A 1 Null)))))))`.

b) Definiți instanțele claselor `Foldable` și `Eq` pentru tipul de date `MList`. Pentru instanța `Foldable`, atât elementele individuale cât și cele stocate în liste Haskell trebuie parcurse de la dreapta la stânga.
Exemplu: `foldr (:) [] (A 3 (L [4, 5] (L [9, 8] (A 1 Null))))` → `[3,4,5,9,8,1]`.
Pentru instanța `Eq`, comparați m-listele aplatizate, utilizând funcția `mFlatten` definită la subpunctul anterior. Puteți folosi funcția `mFlatten` chiar dacă nu ați implementat-o.

c) Implementați funcțiile `mHead` și `mEqual`:

- `mHead` întoarce primul element din m-listă, indiferent dacă apare ca element individual sau într-o listă Haskell. Modelați situația în care m-lista nu conține elemente folosind tipul `Maybe a`.
- `mEqual` verifică dacă toate elementele din structură sunt egale între ele, inclusiv elementele care apar în liste Haskell. Funcția va întoarce `True` pentru o m-listă fără elemente. Implementarea nu poate fi explicit recursivă, și trebuie să utilizeze funcția `all` din Haskell:
`all :: Foldable t => (a -> Bool) -> t a -> Bool`.