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:

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

Soluție: Demonstrație prin inducție structurală după T:
- **Cazul de bază** ($T = \text{Nil}$):

$$\text{MS} = \text{mirror}(\text{fmap}(f, \text{Nil})) = \text{mirror}(\text{Nil}) = \text{Nil}$$

$$\text{MD} = \text{fmap}(f, \text{mirror}(\text{Nil})) = \text{fmap}(f, \text{Nil}) = \text{Nil}$$

- **Pasul de inducție** ($T = \text{Nod}(S, v, D)$):
  Ipoteza inductivă (II):

$$\text{mirror}(\text{fmap}(f, S)) = \text{fmap}(f, \text{mirror}(S)) \quad \text{și} \quad \text{mirror}(\text{fmap}(f, D)) = \text{fmap}(f, \text{mirror}(D))$$

  Demonstrăm pentru T:

$$\begin{aligned}
\text{MS} &= \text{mirror}(\text{fmap}(f, \text{Nod}(S, v, D))) \\
&= \text{mirror}(\text{Nod}(\text{fmap}(f, S), f(v), \text{fmap}(f, D))) \quad \text{[fmap2]} \\
&= \text{Nod}(\text{mirror}(\text{fmap}(f, D)), f(v), \text{mirror}(\text{fmap}(f, S))) \quad \text{[mirror2]} \\
&= \text{Nod}(\text{fmap}(f, \text{mirror}(D)), f(v), \text{fmap}(f, \text{mirror}(S))) \quad \text{[II]}
\end{aligned}$$

$$\begin{aligned}
\text{MD} &= \text{fmap}(f, \text{mirror}(\text{Nod}(S, v, D))) \\
&= \text{fmap}(f, \text{Nod}(\text{mirror}(D), v, \text{mirror}(S))) \quad \text{[mirror2]} \\
&= \text{Nod}(\text{fmap}(f, \text{mirror}(D)), f(v), \text{fmap}(f, \text{mirror}(S))) \quad \text{[fmap2]}
\end{aligned}$$

Barem (10 puncte):
- 1p: Precizarea constructorilor care corespund cazului de bază / pasului de inducție.
- 2p: Cazul de bază ($T = \text{Nil}$).
- 2p: Ipoteza inductivă (pentru subarborii S și D).
- 3p: Pasul de inducție (MS).
- 2p: Pasul de inducție (MD).

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.

Soluție:

1. $(\lambda x.y\ z) \rightarrow_\beta y$ (funcția constantă aplicată pe o variabilă).
2. $(\lambda x.x\ y) \rightarrow_\beta y$ (funcția identitate aplicată pe y).
3. $(\lambda x.y\ \lambda x.x) \rightarrow_\beta y$ (funcția constantă aplicată pe o funcție).

Barem (10 puncte):
- 4p: Prima expresie corectă găsită (oricare ar fi).
- 3px2: Celelalte două expresii.

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
```

Soluție: Varianta corectă este a).
- **Explicație:** `or` din Racket își evaluează argumentele pe rând, oprindu-se la primul `true`. Când apelul `(find 'a '(a c))` întoarce `#t`, întregul `or` se evaluează la `#t`, iar rezultatul se propagă pe stivă, devenind rezultat final.
- **Eroare b):** `or` nu se oprește când primul argument se evaluează la `#t`, ci evaluează și al doilea argument.
- **Eroare c):** Toate apelurile sunt pe stivă. Contraexemplu: al doilea apel al funcției `find` este generat de ramura `else` și este pe coadă.

Barem (10 puncte):
- 2p: Identificarea variantei corecte (a)).
- 4px2: Identificarea și explicarea corectă a erorilor din variantele b), c).

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))`.

Soluție:

```
(define (I n)
  (map (lambda (i)
         (map (lambda (j) (if (= i j) 1 0))
              (range n)))
       (range n)))
```

Barem (10 puncte):
- 3p: Maparea unei funcții pe `(range n)` (sau un `fold` echivalent)).
- 4p: Utilizarea imbricată a unui `map` sau `fold` pe `(range n)` pentru construirea listelor interioare.
- 3p: Logica de comparare a indicilor pentru generarea diagonalei.

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)))
```

Soluție: Codul se evaluează la **3**.
- Zona de vizibilitate a variabilelor definite în `letrec` este întregul `letrec`, așadar expresia `(delay x)` se referă la `x = 2`. Mulțumită evaluării lenețe, variabila x nu este accesată înainte de momentul inițializării.
- În corpul `letrec`-ului, se apelează `(force y)` producând valoarea 2. În punctul definirii funcției `plus-x`, variabila `x` este cea globală, adică `1`. Rezultatul final este `(+ 1 2) = 3`.

Barem (10 puncte):
- 3p: Precizarea faptului că `delay` întârzie evaluarea.
- 3p: Justificarea faptului că funcția `plus-x` folosește `x`-ul global = 1.
- 3p: Justificarea faptului că `y` vede `x = 2`.
- 1p: Rezultatul final.

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, ...], ...]`.

Soluție:

```
streamOfStreams :: [[Int]]
streamOfStreams = loop [0, 1]
  where
    loop s = cycle s : loop ([0] ++ s ++ [1])
```

Barem (10 puncte):
- 3p: Mecanismul de trecere la `n`-ul următor (listă de indici, padding, etc.).
- 3p: Crearea secvenței finite corecte pentru `n`-ul curent.
- 3p: Transformarea fluxurilor interioare în fluxuri infinite.
- 1p: Flux `streamOfStreams` infinit (punctaj condiționat de faptul că se implementează fluxul cerut).

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.

Soluție:

$$\begin{aligned}\text{uncurry} &:: (a \to b \to c) \to (a, b) \to c\\ (.) &:: (e \to f) \to (d \to e) \to d \to f\\ \text{flip} &:: (g \to h \to i) \to h \to g \to i\end{aligned}$$

$e = \text{uncurry} \cdot \text{flip} :: d \to f$ (tipul rezultatului operatorului de compunere), dacă:

1. $e \to f = (a \to b \to c) \to (a, b) \to c$, adică $e = a \to b \to c$ și $f = (a, b) \to c$
2. $d \to e = (g \to h \to i) \to h \to g \to i$, adică $e = h \to g \to i$ și $d = g \to h \to i$

Din unificarea celor două definiții pentru `e`, obținem: $h = a, g = b, i = c$.
$\implies e :: (b \to a \to c) \to (a, b) \to c$

Barem (10 puncte):
- 3p: Specificarea corectă a tipurilor funcțiilor `uncurry`, `flip` și `(.)`.
- 1p: Unificarea tipului funcției `uncurry` cu tipul primului parametru al compunerii.
- 1p: Unificarea tipului funcției `flip` cu tipul celui de-al doilea parametru al compunerii.
- 3p: Rezolvarea constrângerilor.
- 2p: Tip final corect (punctaj condiționat de existența justificărilor).

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]`.

Soluție:

```
data AltList a b = AltNull | AltCons a (AltList b a)

-- Reprezentarea valorii cerute:
altL :: AltList Int Char
altL = AltCons 1 $ AltCons 'a' $ AltCons 2 $ AltCons 'b' $ AltCons 3 AltNull
```

Barem (10 puncte):
- 2p: Definirea unui constructor de tip corect parametrizat.
- 1p: Definirea unui constructor de date pentru lista vidă.
- 3p: Definirea unui constructor de date pentru lista nevidă, asigurând alternanța.
- 2p: Faptul că lista începe cu tipul `a` și se poate termina cu orice tip.
- 2p: Reprezentarea corectă a listei din exemplu.

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`).

Soluție: Rescriem `map length` sub formă de `foldr`:

$$\text{sum . map length} = \text{sum . foldr } (\lambda x\ acc \to \text{length } x : acc)\ []$$

Identificăm componentele pentru a aplica proprietatea de fuziune (h . foldr f a = foldr g b):

$$\begin{aligned}h &= \text{sum}\\ f &= \lambda x\ acc \to \text{length } x : acc\\ a &= []\end{aligned}$$

Utilizăm condițiile din proprietatea de fuziune pentru a determina b și g:
1. $b = h\ a = \text{sum } [] = 0$

2.

$$\begin{aligned} g\ x\ acc = g\ x\ (\text{sum}\ y) &= h\ (f\ x\ y) \\ &= \text{sum}\ ((\lambda x\ acc \to \text{length}\ x : acc)\ x\ y) \\ &= \text{sum}\ (\text{length}\ x : y) \\ &= \text{length}\ x + \text{sum}\ y \\ &= \text{length}\ x + acc \end{aligned}$$

$\implies$ sum . map length = foldr $(\lambda x\ acc \to \text{length}\ x + acc)\ 0$

Barem (10 puncte):

- 2p: Rescrierea cu foldr.
- 1p: Identificarea componentelor.
- 1p: Prima condiție.
- 4p: A doua condiție.
- 2p: Formă finală corectă (punctaj condiționat de existența calculelor).

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.

Soluție:

```
mFlatten :: MList a -> MList a
mFlatten Null = Null
mFlatten (A x rest) = A x (mFlatten rest)
mFlatten (L [] rest) = mFlatten rest
mFlatten (L (x:xs) rest) = A x (mFlatten (L xs rest))
```

Barem (10 puncte): 2p signatura, 2p cazul `Null`, 2p cazul `A`, 4p cazul `L`.

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. Pentru instanța `Eq`, comparați m-listele aplatizate, utilizând funcția `mFlatten` definită la subpunctul anterior.

Soluție:

```
instance Foldable MList where
  foldr _ acc Null = acc
  foldr f acc (A x rest) = f x (foldr f acc rest)
  foldr f acc (L xs rest) = foldr f (foldr f acc rest) xs

instance Eq a => Eq (MList a) where
  l1 == l2 = helper (mFlatten l1) (mFlatten l2)
    where
      helper Null Null = True
      helper (A x rest1) (A y rest2) = x == y && helper rest1 rest2
      helper _ _ = False
```

Barem (10 puncte):

- 5p Foldable - 1p instance cu constructorul de tip, 1p cazul `Null`, 1.5p cazul `A`, 1.5p cazul `L`.
- 5p Eq - 1p constrângerea și instance cu întregul tip, 1p cazul `Null`, 2p cazul `A`, 1p restul.

c) Implementați funcțiile `mHead` și `mEqual` conform descrierilor din cerință (fără recursivitate explicită pentru `mEqual`, utilizând `all`).

Soluție:

```
mHead :: MList a -> Maybe a
mHead Null = Nothing
mHead (A x _) = Just x
mHead (L [] rest) = mHead rest
mHead (L (x:_) _) = Just x

mEqual :: Eq a => MList a -> Bool
mEqual l = case mHead l of
  Nothing -> True
  Just h -> all (== h) l
```

Barem (10 puncte):

- 5p: mHead - 1p signatura, 4p cazurile
- 5p: mEqual - 1p signatura, 2p toate cazurile fără elemente, 2p folosirea funcționalei `all` pentru cazul cu elemente.