Examen Paradigme de Programare
Seria CC — 19.06.2026 — **Varianta B**
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:

$\text{size}(\text{Nil}) = 0$
$\text{size}(\text{Nod}(S, v, D)) = 1 + \text{size}(S) + \text{size}(D)$
$\text{fold}(f, acc, \text{Nil}) = acc$
$\text{fold}(f, acc, \text{Nod}(S, v, D)) = f(\text{fold}(f, acc, S), v, \text{fold}(f, acc, D))$

demonstrați prin inducție structurală proprietatea:

$$\text{fold}(\lambda(s\ v\ d).(1 + s + d),\ 0,\ T) = \text{size}(T), \quad \forall T \in \text{BT}\langle K\rangle$$

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

$$\text{MS} = \text{fold}(\lambda(s\ v\ d).(1 + s + d),\ 0,\ \text{Nil}) = 0$$

$$\text{MD} = \text{size}(\text{Nil}) = 0$$

- **Pasul de inducție** ($T = \text{Nod}(S, v, D)$):
  Ipoteza inductivă (II): Pentru simplitate, notăm $g = \lambda(s\ v\ d).(1 + s + d)$.

$$\text{fold}(g, 0, S) = \text{size}(S) \quad \text{și} \quad \text{fold}(g, 0, D) = \text{size}(D)$$

  Demonstrăm pentru T:

$$\begin{aligned}\text{MS} &= \text{fold}(g, 0, \text{Nod}(S, v, D))\\ &= g(\text{fold}(g, 0, S), v, \text{fold}(g, 0, D)) \quad \text{[fold2]}\\ &= g(\text{size}(S), v, \text{size}(D)) \quad \text{[II]}\\ &= 1 + \text{size}(S) + \text{size}(D) \quad \text{[aplicarea lui g]}\end{aligned}$$

$$\begin{aligned}\text{MD} &= \text{size}(\text{Nod}(S, v, D))\\ &= 1 + \text{size}(S) + \text{size}(D) \quad \text{[size2]}\end{aligned}$$

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: $E = (x\ y)$. Dați 2 λ-expresii diferite (care nu diferă doar prin numele variabilelor) care se reduc în fix 2 pași de beta-reducere la expresia E.

Soluție:

1. $(\lambda z.z\ (\lambda w.w\ (x\ y))) \rightarrow_\beta (\lambda z.z\ (x\ y)) \rightarrow_\beta (x\ y)$ (funcția identitate aplicată de două ori pe (x y)).
2. $((\lambda z.\lambda w.(z\ w)\ x)\ y) \rightarrow_\beta (\lambda w.(x\ w)\ y) \rightarrow_\beta (x\ y)$ (funcție care își aplică primul argument asupra celui de-al doilea, aplicată întâi pe x, apoi pe y).

Barem (10 puncte):
- 5p: Prima expresie corectă găsită.
- 5p: A doua expresie corectă găsită.

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 (all-even? L)
  (cond
    ((null? L) #t)
    ((list? (car L)) (and (all-even? (car L)) (all-even? (cdr L))))
    (else (and (even? (car L)) (all-even? (cdr L))))))
(trace all-even?)
(all-even? '(2 (4 5 6) 8))
```

Varianta a)
```
>(all-even? '(2 (4 5 6) 8))
>(all-even? '((4 5 6) 8))
> (all-even? '(4 5 6))
> (all-even? '(5 6))
< #f
> (all-even? '(8))
> (all-even? '())
< #t
<#f
#f
```

Varianta b)
```
>(all-even? '(2 (4 5 6) 8))
> (all-even? '((4 5 6) 8))
> >(all-even? '(4 5 6))
> > (all-even? '(5 6))
> > >(all-even? '(6))
> > > (all-even? '())
< < < #t
< < <#t
< < #f
< <#f
< #f
<#f
#f
```

Varianta c)
```
>(all-even? '(2 (4 5 6) 8))
>(all-even? '((4 5 6) 8))
> (all-even? '(4 5 6))
> (all-even? '(5 6))
< #f
<#f
#f
```

Soluție: Varianta corectă este c).
- **Explicație:** `and` din Racket își evaluează argumentele pe rând și se oprește la primul `false`. Când apelul `(all-even? '(5 6))` întoarce `#f`, întregul `and` se evaluează la `#f`, iar rezultatul se propagă pe stivă către apelul superior. Întrucât sublista s-a evaluat la `#f`, primul argument al operatorului `and` de pe nivelul superior este `#f`, oprind evaluarea restului listei `'(8)`.
- **Eroare a):** `and` nu se oprește când sublista se evaluează la `#f`, ci evaluează și al doilea argument, apelând funcția pe lista `'(8)`.
- **Eroare b):** `and` nu se oprește când paritatea lui 5 se evaluează la `#f`, ci evaluează și al doilea argument, apelând funcția pe lista `'(6)`.

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

4. Se dă o listă de liste `LL` și un element `x`. În Racket, folosind doar funcționale (fără recursivitate explicită) și fără a aplatiza lista în prealabil, determinați numărul total de apariții ale lui `x` în toate sublistele din `LL`. Sublistele stochează doar elemente individuale.
Exemplu: Pentru lista `'((1 2 3) (1 1) (2 2 3 1 2 3) (4 1))` și elementul 1, rezultatul este 5.

Soluție:

```
(define (count-occurrences LL x)
  (apply + (map (lambda (l)
                  (length (filter (lambda (elem) (equal? elem x)) l)))
```

```
LL)))
```

Barem (10 puncte):
- 3p: Maparea unei funcții pe `LL` (sau un `fold` echivalent).
- 4p: Determinarea numărului de apariții din fiecare sublistă.
- 3p: Calculul numărului total de apariții (suma finală).

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

```
(define x 1)
(define plus-xy (delay (+ x y)))
(let* ((y 0) (x y)) (force plus-xy))
```

Soluție: Evaluarea produce o eroare.
- În contextul global, în care se creează promisiunea `(delay (+ x y))`, există o variabilă `x = 1`, însă nu există nicio variabilă `y`.
- Mulțumită evaluării leneșe, eroarea nu se produce la `define`. Însă în `let*`, `(force plus-xy)` declanșează evaluarea efectivă a expresiei `(+ x y)` în contextul său de definire (cel global), generând o eroare de tip `"unbound identifier"`.

Barem (10 puncte):
- 3p: Precizarea faptului că `delay` capturează contextul din punctul de definire a promisiunii.
- 3p: Precizarea faptului că în corpul `let*`-ului se forțează evaluarea.
- 3p: Menționarea faptului că variabila `y` lipsește din contextul global.
- 1p: Rezultatul final (eroare).

6. În Haskell, definiți un flux de fluxuri în care al n-lea flux repetă la infinit primii n multipli ai lui n.
Notă: Rezultatul este un flux infinit de fluxuri infinite: `[[1,1,1,1,1,1, ...], [2,4,2,4,2,4, ...], [3,6,9,3,6,9, ...], ...]`.

Soluție:

```
streamOfStreams :: [[Int]]
streamOfStreams = map multiples [1 ..]
  where
    multiples n = cycle [i * n | i <- [1 .. n]]
```

Barem (10 puncte):
- 3p: Mecanismul de trecere la `n`-ul următor (listă de indici, recursivitate, 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 = curry . flip`.
Notă: `curry` transformă o funcție binară uncurry într-o funcție echivalentă curry, iar `flip` inversează ordinea celor două argumente ale unei funcții binare.

Soluție:

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

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

1. $e \to f = ((a, b) \to c) \to a \to b \to c$, adică $e = (a, b) \to c$ și $f = a \to b \to c$

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:

$$h \to g \to i = (a, b) \to c \implies h = (a, b), \quad g \to i = c$$

Substituind constrângerile în definițiile pentru d și f:

$$d = g \to (a, b) \to i$$

$$f = a \to b \to g \to i$$

$$\implies e :: (g \to (a, b) \to i) \to a \to b \to g \to i$$

Barem (10 puncte):
- 3p: Specificarea corectă a tipurilor funcțiilor `curry`, `flip` și `(.)`.
- 1p: Unificarea tipului funcției `curry` 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 arbore binar în care nodurile de pe nivelurile pare conțin valori de tip `a`, iar nodurile de pe nivelurile impare conțin valori de tip `b`. Nivelul 0 corespunde rădăcinii, iar arborele poate fi și vid. Folosiți apoi acest tip pentru a reprezenta arborele cu rădăcina 1, fiul stâng `'a'` (fără copii), și fiul drept `'b'` (care are doar un fiu drept - o frunză cu eticheta 2).

Soluție:

```
data AltTree a b = AltNil | AltNode a (AltTree b a) (AltTree b a)

-- Reprezentarea valorii cerute:
altT :: AltTree Int Char
altT = AltNode 1 (AltNode 'a' AltNil AltNil) (AltNode 'b' AltNil (AltNode 2 AltNil AltNil))
```

Barem (10 puncte):
- 2p: Definirea unui constructor de tip corect parametrizat.
- 1p: Definirea unui constructor de date pentru arborele vid.
- 3p: Definirea unui constructor de date pentru arborele nevid, asigurând alternanța.
- 2p: Faptul că rădăcina are tipul `a` și se poate alterna până la orice adâncime.
- 2p: Reprezentarea corectă a arborelui din exemplu.

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

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

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

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

$$h = \text{concat}$$
$$f = \lambda x\ acc \to \text{reverse } x : acc$$
$$a = []$$

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

2.

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

$\implies$ concat . map reverse = foldr $(\lambda x\ acc \to \text{reverse}\ x +\!\!+ acc)$ []

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) Definiți instanțele claselor `Functor` și `Foldable` 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.

Soluție:

```
instance Functor MList where
  fmap _ Null = Null
  fmap f (A x rest) = A (f x) (fmap f rest)
  fmap f (L xs rest) = L (map f xs) (fmap f rest)

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
```

Barem (10 puncte):

- 5p Functor - 1p instance cu constructorul de tip, 1p cazul `Null`, 1.5p cazul `A`, 1.5p cazul `L`.
- 5p Foldable - 1p instance cu constructorul de tip, 1p cazul `Null`, 1.5p cazul `A`, 1.5p cazul `L`.

b) Implementați funcțiile `mHead` și `mMax` conform descrierilor din cerință (fără recursivitate explicită pentru `mMax`, utilizând `maximum`).

Soluție:

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

mMax :: (Ord a) => MList a -> Maybe a
mMax l = case mHead l of
```

```
  Nothing -> Nothing
  _ -> Just $ maximum l
```

Barem (10 puncte):

- 5p: mHead - 1p signatura, 4p cazurile
- 5p: mMax - 1p signatura, 2p toate cazurile fără elemente, 2p folosirea funcției `maximum` pentru cazul cu elemente.

c) Implementați funcția `mReverse` care inversează ordinea elementelor din m-listă. Listele interne (introduse de constructorul `L`) trebuie de asemenea inversate.

Soluție:

```
mReverse :: MList a -> MList a
mReverse l = helper l Null
  where
    helper Null acc = acc
    helper (A x rest) acc = helper rest (A x acc)
    helper (L xs rest) acc = helper rest (L (reverse xs) acc)
```

Barem (10 puncte): 2p signatura, 2p inversarea corectă folosind un acumulator, 1p cazul `Null`, 2p cazul `A`, 3p cazul `L`.