Precizări:

- Fiecare subiect are câte 10p. **Punctajul NU se acordă în absența justificării răspunsului!**
- Este suficientă rezolvarea a **10** subiecte din cele 12 pentru nota maximă.
- Punctajul suplimentar reprezintă **bonus**, ce poate compensa punctajul de pe parcurs.

1. Fie următoarele definiții în calculul lambda:

$$one \equiv \lambda f.f$$
$$increment \equiv \lambda n.\lambda f.(f\ \ (n\ \ f)).$$

Descrieți evaluarea pas cu pas a expresiilor (5p + 5p):

$$(increment\ \ one)$$
$$(increment\ \ (increment\ \ one)).$$

Soluție. .

$$(increment\ \ one) \to \lambda f.(f\ \ (\lambda f.f\ \ f)) \to \lambda f.(f\ \ f)$$
$$(increment\ \ (increment\ \ one)) \to \lambda f.(f\ \ (\lambda f.(f\ \ f)\ \ f)) \to \lambda f.(f\ \ (f\ \ f)).$$ □

2. Fie funcția Racket de mai jos:

```
(define (f L x)
  (if (null? L)
      x
      (+ (car L)
         (f (cdr L) (+ x (car L))))))
```

(a) (5p) Ce calculează `(f L 0)`?

(b) (5p) Ce tip de recursivitate utilizează `f`?

Soluție. .

(a) Dublul sumei elementelor listei `L`, deoarece fiecare element se adună atât pe avans, cât și pe revenire.

(b) Pe stivă, deoarece una din însumările cu `(car L)` se face pe revenire. □

3. Fie următoarea reprezentare în Haskell a unui arbore binar:

```
data Tree a
    = Empty
    | Node a (Tree a) (Tree a)
```

Pornind de la definiția funcționalei `foldr`, rezolvați subpunctele de mai jos:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (h : t) = f h $ foldr f acc t
```

(a) (4p) Care este tipul funcționalei `foldT`, care extinde `foldr` pentru „împăturirea" unui arbore binar?

(b) (4p) Care este implementarea acesteia?

(c) (2p) Pe baza lui `foldT`, definiți o funcție pentru însumarea tuturor cheilor dintr-un arbore numeric.

Soluție. .

```
foldT :: (a -> b -> b -> b) -> b -> Tree a -> b
foldT _ acc Empty = acc
foldT f acc (Node key left right) =
    f key (foldT f acc left) (foldT f acc right)

sumT :: Num a => Tree a -> a
sumT = foldT (\x y z -> x + y + z) 0
```

□

4. Fie următoarea secvență în Racket:

```
(define f
  (lambda (x)
    (lambda (y)
      (+ x y))))

(define x 0)

(define g (f x))
(g 10)

(define x 1)
(g 10)
```

Care sunt valorile celor două aplicații `(g 10)`? Justificați utilizând modelul contextual de evaluare (închideri și contexte).

Soluție. .

Ambele sunt `10`, deoarece, la evaluarea aplicației `(f x)`, valoarea lui `x` din acel moment (`0`) este salvată în închiderea construită: $g \leftarrow \langle \lambda y.(+\ x\ y);\ x \leftarrow 0 \rangle$. □

5. Construiți în Haskell fluxul compunerilor unei funcții cu ea însăși, de ori câte ori. De exemplu, `compositions (+ 1) = [(+ 1), (+ 1) . (+ 1), (+ 1) . (+ 1) . (+ 1), ...]`.

Soluție. .

```
compositions :: (a -> a) -> [a -> a]
compositions f = f : map (f .) (compositions f)
```

□

6. Scrieți în Haskell o funcție care întoarce `True` dacă un element căutat se află într-o listă, eventual **infinită**, folosind funcționala `foldr`. Este acest lucru posibil? Dacă da, definiți-o și explicați de ce funcționează. Dacă nu, justificați de ce nu se poate.

Soluție. .

```
exists :: Eq a => a -> [a] -> Bool
exists element = foldr (||) False . map (== element)
```

Este posibil, deoarece funcția trimisă lui `foldr` poate alege dacă să evalueze sau nu acumulatorul, în baza evaluării leneșe. În cazul de față, acumulatorul este al doilea parametru al lui `(||)`, nemaifiind evaluat dacă primul este `True`. □

7. Sintetizați tipul următoarei funcții Haskell:

```
f x y = [(x, y), (y, x)]
```

Soluție. .

```
f :: a -> b -> [c]
c = (a, b) = (b, a)
a = b
f :: a -> a -> [(a, a)]
```

□

8. Să presupunem că numerele complexe sunt reprezentate în Haskell sub forma unor perechi numerice, e.g. $2 + 3i \equiv$ `(2, 3)`. Instanțiați clasa `Num` cu tipul menționat mai sus, supraîncărcând doar operatorul `+`. De exemplu, `(2, 3) + (20, 30) = (22, 33)`.

```
class Num a where
    (+) :: a -> a -> a
```

Soluție. .

```
instance (Num a, Num b) => Num (a, b) where
    (x1, y1) + (x2, y2) = (x1 + x2, y1 + y2)
```

□

9. Transcrieți în logica cu predicate de ordinul I următoarea propoziție:

   Cine sapă groapa altuia, cade el în ea.

Soluție. .

Utilizăm predicatele

- $sapa(agresor, groapa, victima)$
- $cade(persoana, groapa)$.

$$\forall a.\forall g.(\exists v.sapa(a, g, v) \Rightarrow cade(a, g))$$

□

10. Utilizați rezoluția propozițională pentru a demonstra că propoziția $p \Leftrightarrow \neg p$ este contradictorie. Rezolvările care nu utilizează rezoluția **nu** vor fi punctate!

Soluție. .

Etape:

- Rescriere: $(p \Rightarrow \neg p) \wedge (\neg p \Rightarrow p)$
- Eliminarea implicațiilor: $(\neg p \vee \neg p) \wedge (p \vee p)$
- Forma clauzală: $\{\neg p\}, \{p\}$

- Rezoluție pe cele două clauze: {}

Obținerea clauzei vide indică prezența unei contradicții. □

11. De câte ori se satisface următorul scop în Prolog și care sunt legările aferente?

```
append(X, Y, [1,2,3,4]), member(_, X), member(L, Y), length(Y, L).
```

Soluție. .

Datorită condiției `member(_, X)`, `X` conține cel puțin un element. Situații:

- `X = [1]`, `Y = [2, 3, 4]`
  - `L = 2`, `length([2, 3, 4], 2)`, eșec
  - `L = 3`, `length([2, 3, 4], 3)`, ok
  - `L = 4`, `length([2, 3, 4], 4)`, eșec
- `Y = [3, 4]` sau `Y = [4]` sau `Y = []`
  - 3 și 4 sunt mai mari ca 2, 1, respectiv 0, eșec.

O singură satisfacere. □

12. Scrieți un algoritm Markov care calculează câtul împărțirii la 2 a unui număr natural. Numărul este reprezentat unar, în forma unei secvențe de simboluri `1`, de lungime egală cu numărul. De exemplu, atât pentru numărul 3, reprezentat prin secvența `111`, cât și pentru numărul 2, reprezentat prin secvența `11`, banda va conține la sfârșitul execuției simbolul `1`.

Soluție. .

În programul de mai jos, `a` este o variabilă de lucru.

```
1 Mod2()
2     a11 -> 1a
3     a1 -> a
4     a -> .
5     -> a
6 end
```

Exemplu: `11 -5-> a11 -2-> 1a -4-> 1.`
Exemplu: `111 -5-> a111 -2-> 1a1 -3-> 1a -4-> 1.` □