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. Care este rezultatul aplicației $(\lambda x.(\lambda x.x\ \ x)\ \ a)$, unde

$$a \equiv \lambda y.\lambda z.\lambda s.\lambda w.\lambda t.(((y\ \ t)\ \ (z\ \ w))\ \ s)?$$

*Hint*: Imaginați-vă cum decurge evaluarea conform modelului bazat pe substituție textuală, învățat la Racket.

*Soluție.* .

$(\lambda x.(\lambda x.x\ \ x)\ \ a) \to (\lambda x.x\ \ a) \to a.$ □

2. Se urmărește implementarea în Racket a funcției de concatenare a listelor, `append`, utilizând recursivitate **pe coadă**. Pentru aceasta, fie funcția `f` de mai jos:

```
(define (f X Y)
  (if (null? X) Y
      (f (cdr X) (cons (car X) Y))))
```

   (a) (2p) Implementați funcția de inversare, `reverse`, utilizând **exclusiv** `f`.

   (b) (8p) Implementați funcția de concatenare, `append`, utilizând **exclusiv** `f` și `reverse`.

*Soluție.* .

```
(define (rev A)
  (f A '()))

(define (app1 A B)
  (f (rev A) B))
; sau
(define (app2 A B)
  (rev (f B (rev A))))
```

□

3. Fie următorul tip de arbore oarecare nevid în Haskell:

```
data Tree a = Node
    { key      :: a
    , children :: [Tree a]
    } deriving (Eq, Show)
```

Definiți pentru tipul `Tree a` o funcțională analoagă lui `filter`, care să păstreze dintr-un arbore doar nodurile a căror cheie îndeplinește condiția precizată. Pentru simplitate, rădăcina **NU** va fi verificată. Funcția va avea tipul de mai jos și va utiliza **cel puțin o funcțională standard**:

```
filterTree :: (a -> Bool) -> Tree a -> Tree a
```

*Soluție.* .

```
filterTree f (Node k ch) =
    Node k $ map (filterTree f) $ filter (f . key) ch
```
□

4. Care este valoarea ultimei expresii din codul de mai jos?

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

(define x 10)

(let ([x 20])
  (map (add x) '(1 2 3)))
```

*Soluție.* .

`'(21 22 23)`, întrucât valoarea lui `x` este cea din `let`. □

5. Care sunt valorile expresiilor din liniile 6, 7, 11 și 12, din următorul cod Racket?

```
1  (define x 0)
2
3  (define closure (lambda () x))
4  (define promise (delay x))
5
6  (closure)
7  (force promise)
8
9  (define x 1)
10
11 (closure)
12 (force promise)
```

*Soluție.* .

0, 0, 1, 0. Linia 11 utilizează valoarea calculată în linia 6, la prima forțare (0), în timp ce linia 12 presupune reevaluarea corpului `x`, care acum este 1. □

6. Definiți în Haskell fluxul prefixelor unui alt flux. Spre exemplu, avem:

```
> take 5 $ prefixes [0..]
[[],[0],[0,1],[0,1,2],[0,1,2,3]]
```

*Soluție.* .

```
prefixes :: [a] -> [[a]]
prefixes s = out
  where
    out = [] : zipWith f out s
    f prefix x = prefix ++ [x]
```
□

7. Sintetizați tipul următoarei funcții în Haskell:

```
f g (x, y, z) = map g [x, y, z]
```

*Soluție.* .

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

`x`, `y` și `z` trebuie să aibă același timp, deoarece sfârșesc în aceeași listă. □

8. Supraîncărcați în Haskell operatorul de adunare pentru tipul predefinit

```
data Maybe a = Just a | Nothing
```

Ori de câte ori un operand este `Nothing`, rezultatul va fi `Nothing`.

*Soluție.* .

```
instance Num a => Num (Maybe a) where
    Just x + Just y = Just $ x + y
    _ + _ = Nothing
```

□

9. Care este diferența dintre următoarele expresii logice?

$$\forall x.(P(x) \Rightarrow Q(x)) \tag{1}$$
$$\forall x.P(x) \Rightarrow \forall x.Q(x) \tag{2}$$

*Soluție.* .

Propoziția (1) afirmă că, ori de câte ori proprietatea $P$ este îndeplinită de către un obiect, trebui să fie satisfăcută și proprietatea $Q$ de același obiect. Propoziția (2) afirmă că, în condițiile în care proprietatea $P$ este satisfăcută de *toate* obiectele, același lucru trebuie să se întâmple și cu proprietatea $Q$. □

10. Utilizând metoda rezoluției, demonstați că:

$$\{\exists x.(P(x) \wedge Q(x))\} \models \exists x.P(x).$$

*Soluție.* .

Construim conjuncția între ipoteză și negația concluziei, și o transformăm în FNC:

$$\exists x.(P(x) \wedge Q(x)) \wedge \neg\exists x.P(x)$$
$$\exists x.(P(x) \wedge Q(x)) \wedge \forall x.\neg P(x)$$
$$\exists x.\forall y.(P(x) \wedge Q(x) \wedge \neg P(y))$$
$$P(c_x) \wedge Q(c_x) \wedge \neg P(y)$$
$$\{P(c_x)\}, \{Q(c_x)\}, \{\neg P(y)\}$$

Aplicând rezoluția pentru clauzele $\{P(c_x)\}$ și $\{\neg P(y)\}$, utilizând substituția $\{y \leftarrow c_x\}$, obținem clauza vidă. □

11. Scrieți un predicat Prolog care numără toate aparițiile unui element dintr-o listă. Descrieți semnificația parametrilor.

*Soluție.* .

```
count(_, [], 0).
count(X, [H|T], N) :- count(X, T, N1), (X == H, N is N1 + 1, ! ; N is
    N1).
```
□

12. Scrieți un algortim Markov care, pentru orice secvență alternantă de trei simboluri 0 sau 1, să înlocuiască simbolul din mijloc cu cel din extremități. Spre exemplu, dacă șirul inițial este `010101`, se va obține în final `000111`.

*Soluție.* .

```
Uniform()
    g1g2g1 -> g1g1g1
    -> .
end
```
□