Precizări:

- Primele 9 subiecte au fiecare câte 10p. Cele 3 cerințe ale problemei au fiecare câte 10p. **Punctajul NU se acordă în absența justificării răspunsului!**
- Este suficientă rezolvarea a **10** itemi dintre cei 12 pentru nota maximă.
- Punctajul suplimentar reprezintă **bonus**, ce poate compensa punctajul de pe parcurs.

1. Care sunt expresiile lambda pe care le puteți scrie utilizând exact 4 apariții **legate** ale variabilei x și 2 simboluri λ? Expresiile nu conțin apariții libere ale lui x sau alte variabile.

   Soluție. .

   $\lambda x.\lambda x.(x\ \ x)$, $\lambda x.(\lambda x.x\ \ x)$, $\lambda x.(x\ \ \lambda x.x)$, $(\lambda x.x\ \ \lambda x.x)$ □

   Barem.

   3,33p/expresie. Sunt suficiente 3 expresii.

2. Fie în Racket funcția `deep-length`, care primește o listă ce poate conține alte liste imbricate pe oricâte niveluri, și determină numărul de elemente care nu sunt liste. De exemplu, pentru lista `'(1 (2 3 (4)) 5)` întoarce `5`.

   (a) Implementați funcția utilizând **recursivitate explicită**, cu **cel puțin un apel pe coadă** (*tail call*).

   (b) Câte cadre de stivă utilizează în total implementarea voastră pentru lista dată ca exemplu?

   Soluție. .

   (a) Este acceptată oricare dintre cele două variante:

```racket
(define (deep-length-1 L acc)
  (if (null? L) acc
      (deep-length-1 (cdr L)
                     (+ acc (if (list? (car L))
                                (deep-length-1 (car L) 0)
                                1)))))

(define (deep-length-2 L acc)
  (cond [(null? L) acc]
        [(list? (car L))
         (deep-length-2 (append (car L) (cdr L)) acc)]
        [else
         (deep-length-2 (cdr L) (+ acc 1))]))
```
□

   (b) `deep-length-1` utilizează 3 cadre de stivă, deoarece al doilea apel recursiv nu este pe coadă, și necesită un nou cadru de stivă la întâlnirea unui element listă. `deep-length-2` per se utilizează un singur cadru de stivă, întrucât este recursivă pe coadă. Se pot menționa eventualele cadre adiționale utilizate de `append`, dar nu este necesar.

   Barem.

(a) 8p

- 1p cazul de bază
- 1p distingerea dintre element listă vs. non-listă
- 2p tratare corectă element listă
- 1p tratare corectă element non-listă
- 3p prezența a cel puțin unui apel pe coadă

(b) 2p

- 1p numărul de cadre
- 1p justificare

3. Implementați în Racket, utilizând **exclusiv funcționale**, mai puțin `apply`, funcția `column-sums`, care primește o matrice reprezentată ca o listă de linii, și calculează lista sumelor pe coloane. De exemplu, pentru matricea `'((1 2) (10 20) (100 200))`, rezultatul este `'(111 222)`.

   Soluție. .

```
(define (column-sums matrix)
  (foldr (curry map +)
         (map (lambda (x) 0) (car matrix))
         matrix))
```

□

   Barem.

   - 2p funcționala pentru parcurgerea liniilor (e.g. `foldr`)
   - 3p cazul de bază, linie cu atâtea elemente 0 câte coloane sunt
     - 1p funcționala
     - 1p funcția trimisă funcționalei
     - 1p aplicarea corectă pe parametri
   - 5p combinarea coloanei curente cu lista intermediară a sumelor
     - 2p funcționala
     - 2p funcția trimisă funcționalei
     - 1p aplicarea corectă pe parametri

4. Care ar putea fi fluxul `findme` din definiția de mai jos, astfel încât fluxul `s` să aibă primele 5 elemente $0, 2, -2, 6, -10$?

```
(define s
  (stream-cons 0
    (stream-map (curry * 2)
      (stream-zip-with - findme s))))
```

   Soluție. .

```
        f0       f1       f2       f3       f4      ... (stream-zip-with -)
         0        2       -2        6      -10      ...
        ----------------------------------------------
        f0     f1-2     f2+2     f3-6    f4+10      ... (stream-map (curry * 2))
       -----------------------------------------------
0 |    2f0   2(f1-2)  2(f2+2)  2(f3-6) 2(f4+10)     ... =
0        2       -2        6      -10               ...
```

De aici, se obțin `f0`, `f1`, `f2`, `f3` ca fiind 1, deci `findme` este probabil `ones`. □

Barem.

- 8p ideea rezolvării
  - 1p explicitare simbolică elemente `f`
  - 2p scădere `s`
  - 2p înmulțire elemente cu 2
  - 1p adăugare `0` în față
  - 2p corespondență elemente `s` între cele cunoscute din enunț și cele care rezultă din definție (ultimele 2 rânduri)
- 2p elementele
  - 1,5p calculul câtorva elemente `f`
  - 0,5p concluzia `ones`

5. Sintetizați tipul expresiei Haskell `(:) . (3 :)`.

Soluție. .

```
1  (:) :: a -> [a] -> [a]   -- prima aparitie
2  (:) :: b -> [b] -> [b]   -- a doua aparitie
3  (.) :: (d -> e) -> (c -> d) -> (c -> e)
4  3 :: Num f => f
5  f = b  -- cu constrangerea (Num b), valabila pana la final
6  (3 :) :: [b] -> [b]
7  c -> d = [b] -> [b]
8  c = d = [b]
9  d -> e = a -> [a] -> [a]
10 d = a = [b]
11 e = [a] -> [a] = [[b]] -> [[b]]
12 (:) . (3 :) :: c -> e = Num b => [b] -> [[b]] -> [[b]]
```
□

Barem.

- 1p tip apariție 1 `(:)` (linia 1)
- 1p tip apariție 2 `(:)`, cu variabile de tip diferite (linia 2)
- 1p tip `(.)` (linia 3)
- 2p tip `(3 :)` (liniile 4–6)
- 2p unificare tip parametru 2 `(.)` cu tip `(3 :)` (liniile 7–8)
- 2p unificare tip parametru 1 `(.)` cu tip apariție 1 `(:)` (liniile 9–11)

- 1p tip final (linia 12)

6. Instanțiați în Haskell clasa `Foldable` cu constructorul de tip `Maybe`.

```haskell
class Foldable f where
    foldr :: (a -> b -> b) -> b -> f a -> b

data Maybe a = Just a | Nothing
```

   (a) Care este tipul particularizat al funcționalei `foldr`?

   (b) Scrieți instanța.

   Soluție. .

```haskell
instance Foldable Maybe where
    -- (a -> b -> b) -> b -> Maybe a -> b
    foldr _ b Nothing  = b
    foldr f b (Just a) = f a b
```

□

   Barem.

   (a) 2p

   (b) 8p

   - 1p antet instanță
   - 1p 3 parametri la `foldr`
   - 3p caz `Nothing`
   - 3p caz `Just`

7. Fie traducerea în logica cu predicate de ordinul I a proverbului *Orice naș își are nașul*:

$$\forall n.(\exists o.naș(n, o) \Rightarrow \exists m.naș(m, n)),$$

   unde predicatul $naș(x, y)$ are semnificația că x este nașul lui y. Știind că $naș(Andrei, Bogdan)$, demonstrați prin **reducere la absurd** în tandem cu **rezoluția** că $\exists x.naș(x, Andrei)$.

   Soluție. .

   Transformăm prima propoziție în formă clauzală:

$$\begin{gathered}\forall n.(\neg\exists o.naș(n, o) \vee \exists m.naș(m, n)) \\ \forall n.(\forall o.\neg naș(n, o) \vee \exists m.naș(m, n)) \\ \forall n.\forall o.\exists m.(\neg naș(n, o) \vee naș(m, n)) \\ \forall n.\forall o.(\neg naș(n, o) \vee naș(f(n, o), n)) \\ \neg naș(n, o) \vee naș(f(n, o), n).\end{gathered}$$

   Apoi, transformăm negația concluziei în forma clauzală:

$$\begin{gathered}\neg\exists x.naș(x, Andrei) \\ \forall x.\neg naș(x, Andrei) \\ \neg naș(x, Andrei).\end{gathered}$$

Scriem clauzele care rezultă din ipoteze și negația concluziei (–) și aplicăm rezoluția.

$\{\neg naș(n, o), naș(f(n, o), n)\}$	Ipoteza 1	(1)
$\{naș(Andrei, Bogdan)\}$	Ipoteza 2	(2)
$\{\neg naș(x, Andrei)\}$	Negația concluziei	(3)
$\{naș(f(Andrei, Bogdan), Andrei)\}$	$(1) + (2), n \leftarrow Andrei, o \leftarrow Bogdan$	(4)
$\{\}$	$(3) + (4), x \leftarrow f(Andrei, Bogdan).$	(5)

□

Barem.

- 3p prelucrarea ipotezei 1
  - 0,5p eliminarea implicației
  - 0,5p transformarea cuantificatorului negat
  - 0,5p aducerea cuantificatorilor în față
  - 1p skolemizarea lui m
  - 0,5p clauza (1)
- 0,5p clauza (2)
- 1,5p prelucrarea concluziei negate
  - 0,5p negarea concluziei
  - 0,5p transformarea cuantificatorului negat
  - 0,5p clauza (3)
- 5p rezoluția
  - 3p primul pas de rezoluție
    - ∗ 1p legarea lui n
    - ∗ 1p legarea lui o
    - ∗ 1p clauza (4)
  - 2p al doilea pas de rezoluție
    - ∗ 1p legarea lui x
    - ∗ 1p clauza (5)

8. Definiți în Prolog predicatul `appendN(+L, +N, -Ls)`, care primește o listă `L` și un număr `N`, și leagă `Ls`, pe rând, la câte o listă cu `N` liste **nevide**, care concatenate produc `L`. De exemplu, interogarea `appendN([1, 2, 3], 1, Ls)` produce doar legarea `Ls = [[1, 2, 3]]`, cu `N = 1` listă internă; interogarea `appendN([1, 2, 3], 2, Ls)` produce legările `Ls = [[1], [2, 3]] ; Ls = [[1, 2], [3]]`, ambele cu câte `N = 2` liste interne; interogarea `appendN([1, 2, 3], 3, Ls)` produce doar legarea `Ls = [[1], [2], [3]]`, cu `N = 3` liste interne. ***Hint***: cazul de bază corespunde lui `N = 1`.

Soluție. .

```
appendN(L, 1, [L]) :- !.
appendN(L, N, [L1|Ls]) :-
    L1 = [_|_], L2 = [_|_],
    append(L1, L2, L),
    N1 is N - 1,
    appendN(L2, N1, Ls).
```

□

Barem.

- 1p cazul de bază
- 8p cazul general
  - 2 × 1p subliste nevide
  - 2p descompunerea listei în două
  - 3p descompunerea recursivă a uneia dintre cele două subliste
  - 1p legarea ieșirii
- 1p diferențiere între cazuri, fie prin *cut*, fie prin testarea explicită a lui `N` pe cazul general

9. Definiți în Prolog predicatul `cartProd(+Xss, -Yss)`, care primește o listă de liste `Xss` și leagă `Yss` la produsul cartezian al tuturor listelor interne. De exemplu, interogarea `cartProd([[1, 2], [3], [4, 5]], Yss)` produce legarea `Yss = [[1, 3, 4], [1, 3, 5], [2, 3, 4], [2, 3, 5]]`. Trebuie să utilizați **cel puțin un metapredicat**, dar puteți recurge pe lângă acesta și la recursivitate explicită. ***Hint***: imaginați-vă că ați obținut produsul cartezian al „restului" de liste interne și gândiți-vă ce trebuie să faceți mai departe cu acesta.

   Soluție. .

```
cartProd([], [[]]).
cartProd([Xs|Xss], Zss) :-
    findall(
        [X|Ys],
        (member(X, Xs), cartProd(Xss, Yss), member(Ys, Yss)),
        Zss
    ).
```

□

   Barem.

   - 2p cazul de bază
     - 1p prezența cazului de bază
     - 1p producerea listei cu o listă vidă, și nu a listei vide, întrucât produsul cartezian ar fi întotdeauna vid pentru toate intrările
   - 8p cazul general
     - 2p `findall`
     - 1p extragerea unui element al primei liste
     - 2p determinarea produsului cartezian al restului listelor
     - 1p extragerea unei liste din produsul cartezian al restului listelor
     - 1p construcția unei liste din produsul cartezian al intrării
     - 1p legarea corectă a ieșirii

10. PROBLEMA. Se urmărește reprezentarea în Haskell a unor propoziții (restricționate) din **logica propozițională**, utilizând următoarele tipuri de date:

```
data Prop
    = Simple Name
    | Neg Prop
    | And [Prop]
    | Or [Prop]
    | Implies Prop Prop

type Name    = Char
type Interpr = [(Name, Bool)]
```

Astfel, o *propoziție* poate fi una simplă (p, q etc.), o negație, conjuncția sau disjuncția mai multor propoziții, sau o implicație. De exemplu, propoziția $p \vee q$ se reprezintă prin `prop = Or [Simple 'p', Simple 'q']`. Așa cum știți de la curs, o *interpretare* este o listă de asocieri între propozițiile simple și valori de adevăr; de exemplu, `interpr = [('p', True), ('q', False)]`.

(a) Definiți funcția `evaluate :: Interpr -> Prop -> Bool`, care evaluează o propoziție sub o anumită interpretare. Folosiți cât mai mult funcționale.

(b) Definiți funcția `truthTable :: [Name] -> [Interpr]`, care construiește un tabel de adevăr, înțeles ca lista tuturor interpretărilor posibile, câte una pe linie, pornind de la numele propozițiilor simple. Pentru punctaj maxim, **evitați recursivitatea explicită**, și utilizați în schimb funcționale și/sau *list comprehensions*!

(c) Definiți funcția `derivesFrom :: [Name] -> Prop -> [Prop] -> Bool`, astfel încât aplicația `derivesFrom names conclusion premises` întoarce `True` dacă concluzia derivă logic din lista premiselor, `names` fiind lista de nume ale propozițiilor simple, i.e. dacă toate interpretările care satisfac simultan toate premisele satisfac și concluzia. Pentru punctaj maxim, **evitați recursivitatea explicită**, și utilizați în schimb funcționale și/sau *list comprehensions*!

Exemple:

- `evaluate interpr prop` → `True`
  (parametri definiți în enunț)
- `truthTable ['p']` →
  ```
  [[('p', False)],
   [('p', True )]]
  ```
- `truthTable ['p', 'q']` →
  ```
  [[('p', False), ('q', False)],
   [('p', False), ('q', True )],
   [('p', True ), ('q', False)],
   [('p', True ), ('q', True )]]
  ```
- ```
  derivesFrom ['p', 'q']
              (Simple 'q')
              [Implies (Simple 'p')
                       (Simple 'q'),
               Simple 'p']
  ```
  → `True` (regula *modus ponens*)
  ```

- derivesFrom ['p', 'q']
  (Simple 'p')
  [Implies (Simple 'p')
  (Simple 'q'),
  Simple 'q']

  → False

Soluție. .

```haskell
data Prop
    = Simple Name
    | Neg Prop
    | And [Prop]
    | Or [Prop]
    | Implies Prop Prop

type Name    = Char
type Interpr = [(Name, Bool)]

prop = Or [Simple 'p', Simple 'q']
interpr = [('p', True), ('q', False)]

evaluate :: Interpr -> Prop -> Bool
evaluate interpr prop = case prop of
    Simple name   -> foldr (\(name', value) acc ->
                              if name == name' then value else acc)
                          False
                          interpr
    Neg p         -> not $ eval p
    And ps        -> and $ map eval ps
    Or ps         -> or $ map eval ps
    Implies p1 p2 -> not (eval p1 && not (eval p2))
  where
    eval = evaluate interpr

truthTable :: [Name] -> [Interpr]
truthTable = foldr f [[]]
  where
    f name acc = [(name, v) : row | v <- [False, True], row <- acc]

derivesFrom :: [Name] -> Prop -> [Prop] -> Bool
derivesFrom names conclusion premises = and relevantValues
  where
    relevantValues = [eval conclusion | interpr <- truthTable names,
                                        let eval = evaluate interpr,
                                        and (map eval premises)]
```

□

Barem.

(a) 5×2p/constructor de date, cu depunctare de câte 1p dacă nu s-au utilizat funcționale la Simple, And, Or.

(b)
- 1p cazul de bază
- 2p extragerea unei linii din tabelul construit recursiv
- 2p extragerea unei valori curente de adevăr
- 2p asamblarea unei linii din tabelul original, cu etichetarea corectă
- 3p absența recursivității explicite

(c)
- 2p construcția tabelului de adevăr
- 1p extragerea unei interpretări din acesta
- 2p evaluarea tututor premiselor în interpretarea curentă
- 1p păstrarea interpretărilor care satisfac simultan toate premisele
- 1p verificarea satisfacerii concluziei în toate interpretările păstrate
- 3p absența recursivității explicite