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. Cu ce se pot înlocui punctele din expresia de mai jos, astfel încât aceasta să se evalueze la z?

$$(\lambda x.(x \;\; y) \;\; \ldots) \to z$$

Soluție. .

Dacă notăm punctele cu E, obținem $(\lambda x.(x \;\; y) \;\; E) \to (E \;\; y)$. Rezultă că E trebuie să fie o funcție, dar, din moment ce y nu apare în rezultatul dorit, funcția trebuie să fie constantă, întorcându-l pe z: $E \equiv \lambda x.z$. □

Barem.

- 3p primul pas de evaluare
- 4p justificare funcție constantă
- 3p forma finală

2. Fie următoarea funcție în Racket:

```
(define (f n x L)
  (if (null? L)
      L
      (if (equal? (car L) x)
          (if (= n 1)
              (cdr L)
              (cons (car L)
                    (f (sub1 n) x (cdr L))))
          (cons (car L)
                (f n x (cdr L))))))
```

(a) (5p) Ce calculează funcția `f` și ce tip de recursivitate utilizează?

(b) (5p) Rescrieți funcția utilizând celălalt tip de recursivitate și numiți-l.

Soluție. .

(a) Elimină a `n`-a apariție a lui `x` din `L`. Utilizează recursivitate pe stivă.

(b) Recursivitate pe coadă:

```
(define (f2 n x L acc)
  (if (null? L)
      (reverse acc)
      (if (equal? (car L) x)
          (if (= n 1)
              (append (reverse acc) (cdr L))
              (f2 (sub1 n) x (cdr L) (cons (car L) acc)))
          (f2 n x (cdr L) (cons (car L) acc)))))
```

□

Barem.

(a) 5p
- 3p funcționalitate
- 2p tip recursivitate

(b) 5p
- 1p `reverse acc`
- 1p `append (reverse acc) (cdr L)`
- 1p prima aplicație recursivă
- 1p a doua aplicație recursivă
- 1p tip recursivitate

3. Definiți în Racket funcția `mean`, care calculează media elementelor unei liste, utilizând **exclusiv funcționale**, și parcurgând lista **o singură dată**. Utilizați și o formă de `let` pentru **evitarea calculelor duplicate**.

Soluție. .

```
(define (mean L)
  (let ([pair (foldl (lambda (x acc)
                       (cons (+ (car acc) x) (+ (cdr acc) 1)))
                     '(0 . 0)
                     L)])
    (/ (car pair) (cdr pair))))
```
□

Barem.

- 6p exclusiv funcționale (0p pt recursivitate explicită)
  - 2p funcțională corectă
  - 1p caz de bază
  - 3p cazul general
- 2p o singură trecere
- 2p evitare calcule duplicate

4. Determinați primele 8 elemente ale fluxului `s` din Racket:

```
(define s
  (stream-cons 1
    (stream-cons 2
      (stream-zip-with - s naturals))))
```

Soluție. .

Scădem `naturals` din `s`, iar dacă adăugăm `1` și `2` în fața rezultatului, obținem `s`:

```
          s0   s1   s2   s3   s4   s5   ... (stream-zip-with -)
           0    1    2    3    4    5   ...
        ------------------------------------
 1  2 | s0-0 s1-1 s2-2 s3-3 s4-4 s5-5   ... =
s0 s1     s2   s3   s4   s5   s6   s7   ... =
 1  2      1    1   -1   -2   -5   -7
```

□

Barem.

- 6p ideea rezolvării
  - 1p explicitare simbolică elemente `s`
  - 2p scădere naturals
  - 1p adăugare `1` și `2`
  - 2p corespondență elemente între `s` și definiția lui `s`
- 4p elementele (0,5p × 8)

5. Sintetizați tipul expresiei Haskell `(map map)`.

Soluție. .

```
1 map :: (a -> b) -> [a] -> [b]  -- prima aparitie
2 map :: (c -> d) -> [c] -> [d]  -- a doua aparitie
3 a -> b = (c -> d) -> [c] -> [d]
4 a = c -> d
5 b = [c] -> [d]
6 map map :: [c -> d] -> [[c] -> [d]]
```

□

Barem.

Câte 2p pt fiecare linie, cu excepția liniei 3, opțională.

6. În Haskell, tipul operatorului de compunere este `(.) :: (b -> c) -> (a -> b) -> (a -> c)`. Ne propunem să generalizăm acest operator, astfel încât tipul rezultatului întors să fie împachetat într-un constructor de tip unar:

```
1 class Composable t where
2     compose :: (b -> t c)
3             -> (a -> t b)
4             -> (a -> t c)
```

(a) (2p) Dacă dorim să instanțiem clasa de mai sus cu constructorul de tip `Maybe`, care ar fi tipul particularizat al lui `compose`?

(b) (8p) Scrieți instanța pentru `Maybe` de mai sus.

Soluție. .

```
1 instance Composable Maybe where
2     -- (b -> Maybe c) -> (a -> Maybe b) -> (a -> Maybe c)
3     compose f g a = case g a of
4         Nothing -> Nothing
5         Just b  -> f b
```

□

Barem.

(a) 2p

- 0,5p structura generală a tipului

- 0,5p × 3 pt fiecare tip intern de funcție

(b) 8p

- 1p antet instanță
- 1p parametri `compose`
- 1p aplicare `(g a)`
- 1p testare rezultat `(g a)`
- 2p caz `Nothing`
- 2p caz `Just`

7. Din punct de vedere logic, propoziția $\forall x.(P(x) \vee Q(x))$ **NU** implică propoziția $\forall y.P(y) \vee \forall z.Q(z)$ (gândiți-vă la numere naturale, $P \equiv$ *par* și $Q \equiv$ *impar*, i.e. faptul că orice număr natural e fie par, fie impar, nu implică faptul că fie toate numerele sunt pare, fie că toate sunt impare). Să presupunem că încercăm totuși să o demonstrăm pe a doua din prima utilizând **reducerea la absurd** în tandem cu **rezoluția**. În ce punct eșuează procesul de demonstrare?

   Soluție. .

   Propoziția $\forall x.(P(x) \vee Q(x))$ are forma clauzală $\{P(x), Q(x)\}$ (1). Negând presupusa concluzie, obținem:

$$\neg(\forall y.P(y) \vee \forall z.Q(z))$$
$$\exists y.\neg P(y) \wedge \exists z.\neg Q(z)$$
$$\exists y.\exists z.(\neg P(y) \wedge \neg Q(z))$$
$$\neg P(c) \wedge \neg Q(d)$$

   Se obțin clauzele $\{\neg P(c)\}$ (2) și $\{\neg Q(d)\}$ (3), unde c și d sunt constante obținute prin skolemizare. Rezolvând clauzele (1) și (2), cu legarea $x \leftarrow c$, obținem clauza $\{Q(c)\}$ (4). Din păcate, nu putem rezolva mai departe clauzele (3) și (4), din cauză că nu putem unifica constantele c și d. □

   Barem.

   - 1p clauza (1)
   - 1p negarea concluziei
   - 1p introducerea negației în paranteze
   - 1p transformarea cuantificatorilor universali în existențiali prin negație
   - 1p cele două skolemizări
   - 1p clauzele (2) și (3)
   - 2p prima aplicare a rezoluției, cu legarea aferentă
   - 2p eșecul celei de-a doua aplicări, cu justificare

8. Se urmărește generarea în Prolog a combinărilor de K elemente dintr-o listă de N elemente, cu $K \leq N$.

   (a) (5p) Definiți predicatul `extract(?Elem, +List, ?Rest)`, care leagă parametrul `Elem` la un element al listei `List`, și parametrul `Rest`, la lista elementelor din dreapta lui. De exemplu, interogarea `extract(Elem, [1, 2], List)` produce legările `Elem = 1, List = [2]`, respectiv `Elem = 2, List = []`.

(b) (5p) Utilizând `extract`, definiți predicatul `combs(+K, +List, ?Comb)`, care leagă parametrul `Comb` la o combinare de `K` elemente ale listei `List`. De exemplu, interogarea `combs(2, [a, b, c], Comb)` produce legările `Comb = [a, b] ; Comb = [a, c] ; Comb = [b, c]`.

Soluție. .

```
extract(X, [X|L], L).
extract(X, [_|T], R) :- extract(X, T, R).

combs(0, _, []) :- !.
combs(K, L, [X|C]) :- extract(X, L, M), K1 is K - 1, combs(K1, M, C).
```

□

Barem.

(a) 5p
- 2p cazul cu element pe prima poziție în listă
  - 1p legare element
  - 1p legare rest
- 3p cazul cu element în interiorul listei
  - 1p aplicație recursivă
  - 1p legare element
  - 1p legare rest

(b) 5p
- 0,5p cazul de bază
- 4p cazul general
  - 1p aplicație `extract`
  - 1p calcul nou `K`
  - 1p aplicație recursivă
  - 1p legare combinare
- 0,5p diferențiere corectă între cazuri, fie prin `cut`, fie cu testare explicită a condiției `K > 0` în al doilea caz

9. Pornind de la o listă de numere, `Xs`, și de la o listă de liste de numere, `Yss`, scrieți în Prolog o interogare care determină lista tuturor listelor din `Yss` care fie au toate elementele în `Xs`, fie nu au niciun element în `Xs`. De exemplu, dacă `Xs = [1, 2, 3]` și `Yss = [[1, 4], [1, 3, 2], [4]]`, rezultatul este lista `[[1, 3, 2], [4]]`. **Atenție!** NU folosiți recursivitate explicită, ci **metapredicate**!

Soluție. .

```
findall( Ys,
          ( member(Ys, Yss),
            ( forall(member(Y, Ys), member(Y, Xs));
              forall(member(Y, Ys), \+ member(Y, Xs))
            )
          ),
          Zss
        ).
```

□

Barem.

- 2p enumerare liste din `Yss`
- 3p condiția apartenenței tuturor elementelor unei liste individuale la `Xs`
- 3p condiția neapartenenței vreunui element al unei liste individuale la `Xs`
- 1p *și* între `member` și `forall`-uri
- 1p *sau* între `forall`-uri

0p pt recursivitate explicită

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

```
data Term = Const String | Var String
    deriving Eq
data Sentence = Atom String [Term]
type Substitution = [(String, Term)]
```

Astfel, un *termen* este o constantă sau o variabilă, iar o *propoziție* este un atom (aplicația unui predicat pe niște termeni). De exemplu, utilizând notația din Prolog, atomul $p(X, c)$, care corespunde aplicației predicatului p asupra variabilei X și constantei c, se poate reprezenta prin `Atom "p" [Var "X", Const "c"]`. O *substituție* este o mulțime de legări de nume de variabile la termeni. De exemplu, substituția $\{X \leftarrow Y, Y \leftarrow c\}$ se poate reprezenta ca `subst = [("X", Var "Y"), ("Y", Const "c")]`. Pentru punctaj maxim, toate funcțiile de mai jos trebuie implementate **fără recursivitate explicită**.

(a) Definiți funcția `lookup :: Substitution -> Term -> Term`, care caută o variabilă într-o subtituție și întoarce termenul aferent din pereche, dacă ea este legată. Pentru variabile nelegate sau pentru constante, funcția întoarce termenul primit ca parametru. De asemenea, definiți funcția `lookupIterate :: Substitution -> Term -> [Term]`, care aplică la infinit funcția `lookup` pe rezultatul căutării anterioare.

(b) Definiți funcția `lookupEnd :: Substitution -> Term -> Term`, care întoarce ultimul element diferit de anterioarele din lista întoarsă de `lookupIterate`.

(c) Definiți funcția `unifyTerms :: Substitution -> Term -> Term -> Bool`, care verifică dacă doi termeni unifică, în baza substituției primite ca parametru. De asemenea, definiți funcția `unifySentences :: Substitution -> Sentence -> Sentence -> Bool`, care verifică dacă doi atomi unifică în baza substituției. Trebuie să verificați numele atomilor și termenii de pe aceleași poziții.

Exemple:

- `lookup subst (Var "X")` → `Var "Y"`
- `lookup subst (Var "Z")` → `Var "Z"`
- `lookup subst (Const "c")` → `Const "c"`
- `take 4 $ lookupIterate subst (Var "X")` → `[Var "X", Var "Y", Const "c", Const "c"]`
- `lookupEnd subst (Var "X")` → `Const "c"`
- `unifyTerms subst (Var "X") (Var "Y")` → `True`

- `unifyTerms subst (Var "X") (Const "c")` → `True`
- `unifyTerms subst (Var "X") (Const "d")` → `False`
- `unifyTerms subst (Const "c") (Const "d")` → `False`
- `unifySentences subst (Atom "p" [Var "X", Const "c"]) (Atom "p" [Var "Y", Var "X"])` → `True`
- `unifySentences subst (Atom "p" [Var "X", Const "c"]) (Atom "u" [Var "Y", Var "X"])` → `False`
- `unifySentences subst (Atom "p" [Var "X", Const "c"]) (Atom "p" [Var "Y", Const "d"])` → `False`

Soluție. .

```haskell
data Term
    = Const String
    | Var String
    deriving (Eq, Show)

data Sentence = Atom String [Term]
    deriving Show

type Substitution = [(String, Term)]

subst :: Substitution
subst = [("X", Var "Y"), ("Y", Const "c")]

lookup :: Substitution -> Term -> Term
lookup subst var@(Var name) = foldr f var subst
  where
    f (name', term) found = if name == name' then term else found
lookup _ term = term

lookupIterate :: Substitution -> Term -> [Term]
lookupIterate subst term = iterate (lookup subst) term

lookupEnd :: Substitution -> Term -> Term
lookupEnd subst term = fst $ head $ filter (\(t1, t2) -> t1 == t2) $
    zip result $ tail result
  where
    result = lookupIterate subst term

unifyTerms :: Substitution -> Term -> Term -> Bool
unifyTerms subst term1 term2 =
    lookupEnd subst term1 == lookupEnd subst term2

unifySentences :: Substitution -> Sentence -> Sentence -> Bool
unifySentences subst (Atom name1 terms1) (Atom name2 terms2) =
    name1 == name2 && length terms1 == length terms2 &&
    and (zipWith (unifyTerms subst) terms1 terms2)
```

□

Barem.

(a)
- 8p `lookup`
  - 3p absența recusivității explicite
  - 1p parcurgere substituție
  - 1p localizare variabilă legată
  - 1p extragere termen aferent
  - 1p caz variabilă nelegată
  - 1p caz constantă
- 2p `lookupIterate`
  - 2p absența recusivității explicite

(b)
- 3p absența recursivității explicite
- 2p parcurgere rezultat `lookupIterate`
- 3p determinare ultim element diferit
- 2p întoarcere element

(c)
- 2p `unifyTerms`
  - 1p aplicare `lookupEnd` pe cei doi termeni
  - 1p verificare egalitate
- 8p `unifySentences`
  - 3p absența recusivității explicite
  - 1p verificare nume atomi
  - 2p aplicare `unifyTerms` pe fiecare pereche de termeni
  - 2p îmbinare rezultate unificare pentru toți termenii