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 din cei 12 pentru nota maximă.
- Punctajul suplimentar reprezintă **bonus**, ce poate compensa punctajul de pe parcurs.

1. Fie λ-expresia de mai jos:

$$E \equiv (\lambda x.\lambda y.(x\ \ y)\ \ y)$$

   (a) (4p) Completați fraza: Pentru a evalua expresia utilizând modelul bazat pe **substituție textuală**, trebuie înlocuite aparițiile (libere/legate?) _(1)_ ale variabilei _(2)_ din expresia _(3)_ cu expresia _(4)_.

   (b) (3p) Care este rezultatul evaluării și de ce nu este corect?

   (c) (3p) Ce pas lipsește pentru a obține rezultatul corect? Scrieți rezultatul corect.

   Soluție. .

   (a) (1) = libere, (2) = x, (3) = $\lambda y.(x\ \ y)$, (4) = y.

   (b) Evaluarea mecanică ar duce la rezultatul $\lambda y.(y\ \ y)$, eronat, întrucât apariția lui y din dreapta lui E își schimbă starea din liberă în legată.

   (c) Trebuie redenumită apariția lui y din stânga lui E în z, obținând $\lambda z.(y\ \ z)$, cu conservarea stării lui y. □

   Barem.

   (a) • 4 × 1p/completare corectă

   (b) • 1p rezultatul greșit
   • 2p justificare

   (c) • 1p rezultatul corect
   • 2p justificare

2. Scrieți în Racket o funcție **recursivă pe stivă** care să compună funcția `f` cu ea însăși de `n` ori, întorcând funcția rezultantă. De exemplu, `((multicompose 5 add1) 0)` se evaluează la `5`. **Atenție!** Folosiți o formă de `let` astfel încât funcția `f` să **NU** fie trimisă ca parametru la fiecare aplicație recursivă.

   Soluție. .

```
1 (define (multicompose n f)
2   (let loop ([n n])
3     (if (zero? n)
4         (lambda (x) x)
5         (compose f (loop (- n 1))))))
```

□

   Barem.

   • 7p funcționare corectă

  - 1p condiție caz de bază
  - 2p rezultat caz de bază
  - 4p aplicație recursivă
- 3p `named let`
  - 1p parametru
  - 2p evitarea trimiterii lui `f` ca parametru la aplicația recursivă

3. Scrieți în Racket, folosind **exclusiv funcționale**, o funcție care ia ca parametri o listă de funcții și o listă de argumente, posibil de lungimi diferite, și aplică fiecare funcție pe fiecare argument, pe principiul unui produs cartezian, întorcând lista rezultatelor. De exemplu, `(cartesian-app (list add1 sub1 odd?) (list 1 2))` se evaluează la `'(2 3 0 1 #t #f)`. Primele două elemente ale rezultatului, `2` și `3`, corespund aplicațiilor `(add1 1)` și `(add1 2)`; următoarele două, `0` și `1`, aplicațiilor `(sub1 1)` și `(sub1 2)` etc.

   Soluție. .

```
(define (cartesian-app fs xs)
  (apply append
         (map (lambda (f)
                (map (lambda (x) (f x)) xs))
              fs)))
```

□

   Barem.

   - 3p parcurgere listă de funcții cu `map` sau `fold`
   - 3p parcurgere listă de argumente cu `map` sau `fold`
   - 1p aplicare funcție individuală pe argument individual
   - 3p aplatizare listă

4. Să presupunem că în Racket există funcția `stream-merge`, care primește două fluxuri pe care le îmbină alternându-le elementele. De exemplu, `(stream-merge (stream 0 2 4 ...) (stream 1 3 5 ...))` se evaluează la `(stream 0 1 2 3 4 5 ...)`, adică la `naturals`.

   Determinați primele 8 elemente ale fluxului `s`:

```
(define s
  (stream-cons 'x (stream-merge naturals s)))
```

   Soluție. .

   Îmbinăm fluxurile `naturals` și `s`, iar dacă adăugăm `'x` în fața rezultatului, obținem `s`:

```
    0  1  2  3  ...              (stream-merge)
    s0 s1 s2 s3 ...
    ---------------------------
x | 0  s0 1  s1 2  s2 3  s3 ... =
s0  s1 s2 s3 s4 s5 s6 s7    ... =
x   0  x  1  0  2  x  3
```

□

Barem.

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

5. Sintetizați tipul următoarei funcții în Haskell:

```
1 f xs = head xs xs
```

Soluție. .

```
1 f :: a -> b
2 xs :: a
3 head :: [c] -> c
4 a = [c]
5 head xs :: c = d -> e
6 d = [c]
7 c = [c] -> e
```

Eroare de tip, întrucât s-ar obține un tip infinit, dacă `c` s-ar lega la o expresie de tip care îl conține strict. □

Barem.

- 1p tip inițial `f` și `xs`
- 1p tip `head`
- 1p unificare tip parametru formal/actual `head` (linia 4)
- 1p tip `head xs`
- 1p unificare tip parametru formal/actual `head xs` (linia 6)
- 2p unificare ciclică (linia 7)
- 3p conchidere eroare

6. Generalizați funcționala `filter` din Haskell, având tipul `(a -> Bool) -> [a] -> [a]`, pentru a opera pe orice constructor de tip unar ce utilizează valori de tipul `a`, nu numai pe constructorul listă.

   (a) (3p) **Definiți** clasa `Filterable`, parametrizată cum considerați de cuviință, în care să definiți funcția `filter'`, cu tipul adecvat.

   (b) (2p) **Instanțiați** `Filterable`, definită mai sus, cu constructorul listă standard.

   (c) (5p) **Instanțiați** `Filterable`, definită mai sus, cu constructorul de tip `Maybe`, definit prin

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

Soluție. .

```haskell
class Filterable t where
    filter' :: (a -> Bool) -> t a -> t a

instance Filterable [] where
    filter' = filter

instance Filterable Maybe where
    filter' _ Nothing = Nothing
    filter' f j@(Just a) = if f a then j else Nothing
```
□

Barem.

(a)
- 1p antet
- 2p aplicare corectă constructor de tip pe variabilă de tip, `t a`

(b)
- 1p antet, instanțiere cu `[]`, nu cu `[a]`
- 1p definiție `filter'`

(c)
- 1p antet, instanțiere cu `Maybe`, nu cu `Maybe a`
- 1p caz `Nothing`
- 1p caz `Just`, condiție
- 1p caz `Just`, rezultat pt satisfacere condiție
- 1p caz `Just`, rezultat pt nesatisfacere condiție

7. Demonstrați utilizând rezoluția și reducerea la absurd că propoziția

$$\forall x.(P(x) \Rightarrow Q(x))$$

are drept consecință logică propoziția

$$\exists y.P(y) \Rightarrow \exists z.Q(z).$$

(a) (2p) Aduceți prima propoziție la forma clauzală.

(b) (5p) Negați a doua propoziție și aduceți rezultatul la forma clauzală.

(c) (3p) Aplicați rezoluția pe clauzele obținute anterior.

Soluție. .

(a) Rescriem implicația și eliminăm cuantificatorul universal, și obținem $\neg P(x) \vee Q(x)$, corespunzând clauzei $\{\neg P(x), Q(x)\}$ (1).

(b) Negăm concluzia și transformăm în forma clauzală urmând pașii:

$$\begin{aligned}
&\neg(\exists y.P(y) \Rightarrow \exists z.Q(z))\\
&\neg(\neg\exists y.P(y) \vee \exists z.Q(z))\\
&\exists y.P(y) \wedge \neg\exists z.Q(z)\\
&\exists y.P(y) \wedge \forall z.\neg Q(z)\\
&P(c_y) \wedge \neg Q(z)
\end{aligned}$$

Se obțin clauzele $\{P(c_y)\}$ (2) și $\{\neg Q(z)\}$ (3), unde c_y este constanta obținută prin skolemizare.

(c) Aplicând rezoluția pe clauzele (1) și (2), cu legarea $x \leftarrow c_y$, obținem clauza $\{Q(c_y)\}$ (4). Aplicând din nou rezoluția pe clauzele (3) și (4), cu legarea $z \leftarrow c_y$, obținem clauza vidă. □

Barem.

(a)
- 1p eliminarea implicației
- 1p scrierea clauzei

(b)
- 1p eliminarea implicației
- 1p introducerea negației în paranteze
- 1p transformarea cuantificatorului existențial în universal prin negație
- 1p eliminarea cuantificatorilor existențial și universal
- 1p scrierea clauzei

(c)
- 1p prima aplicare a rezoluției
- 0,5p legarea lui x
- 1p a doua aplicare a rezoluției
- 0,5p legarea lui z

8. Definiți în Prolog predicatul `group(+Xs, -Yss)`, care primește o listă de numere nenule `Xs`, și leagă lista de liste `Yss`, obținute prin gruparea elementelor consecutive cu același semn. De exemplu, interogarea `group([1, 2, 3, -4, -5, 6], Yss)` produce legarea `Yss = [[1, 2, 3], [-4, -5], [6]]`.

Soluție. .

```
group([], []).
% un singur element
group([X], [[X]]).
% cel putin 2 elemente, avand acelasi semn
group([X,Y|Zs], [[X|Rs]|Rss]) :- X*Y > 0, !, group([Y|Zs], [Rs|Rss]).
% cel putin 2 elemente, avand semn diferit
group([X,Y|T], [[X]|Rss]) :- group([Y|T], Rss).
```
□

Barem.

- 1p cazul de bază pt lista vidă
- 1p cazul de bază pt lista cu un element
- 3p cazul cu același semn, adăugare la o listă interioară existentă
- 3p cazul cu semne diferite, introducere listă interioară nouă
- 2p diferențierea corectă a ultimelor două cazuri, fie cu `cut`, fie cu explicitarea condiției pe ambele cazuri

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 conțin cel puțin două dintre elementele lui `Xs`. De exemplu, dacă `Xs = [1, 2, 3]` și `Yss = [[1], [1, 2], [1, 3], [1, 3, 4], [1, 4, 5]]`, rezultatul este lista `[[1, 2], [1, 3], [1, 3, 4]]`. **Atenție!** NU folosiți recursivitate explicită, ci **metapredicate**!

Soluție. .

```
findall(Ys,
        (member(Ys, Yss),
         findall(X,
                 (member(X, Xs), member(X, Ys)),
                 [_,_|_])),
        Zss)
```
□

Barem.

- 2p parcurgere `Yss`
- 2p parcurgere `Xs`
- 2p verificare apartenență element din `Xs` la listă individuală din `Yss`
- 2p condiția de minim 2 elemente
- 2p unică satisfacere pentru întreaga listă de liste, nu satisfaceri multiple pt liste individuale

10. PROBLEMA. Se urmărește reprezentarea în Haskell a clauzelor din **logica propozițională** (fără predicate, variabile sau cuantificatori), utilizând următoarele definiții de tipuri de date:

```
data Literal = Positive String
             | Negative String
data Clause = Clause [Literal]
```

De exemplu, clauza $\{p, \neg q, r\}$, care corespunde disjuncției $p \vee \neg q \vee r$, se poate reprezenta prin `Clause [Positive "p", Negative "q", Positive "r"]`.

(a) Instanțiați clasa `Show` cu tipurile `Literal` și `Clause`, astfel încât aplicația lui `show` pe clauza din exemplul de mai sus să se evalueze la șirul `"p v ~q v r"`. **Atenție!** NU utilizați funcții ca `intersperse` sau `intercalate`.

(b) Definiți funcția `decompose`, având tipul `[a] -> [(a, [a])]`, care primește o listă și elimină, pe rând, câte un singur element la un moment dat, întorcându-l în pereche cu restul listei, fără acesta. De exemplu, `decompose [1,2,3]` produce `[(1,[2,3]),(2,[1,3]),(3,[1,2])]`.

(c) Pe baza funcției `decompose`, și a funcției deja definite `complementary :: Literal -> Literal -> Bool`, care verifică dacă doi literali sunt complementari (de exemplu, p și $\neg p$), scrieți funcția `resolve :: Clause -> Clause -> Maybe Clause`, care rezolvă două clauze pe baza primei perechi de literali complementari, câte unul din fiecare clauză, dacă aceasta există. *Hint*: *list comprehensions*. Exemple:

- `resolve (Clause [Positive "p"]) (Clause [Negative "q"])` → `Nothing`
- `resolve (Clause [Positive "p"]) (Clause [Negative "p", Negative "q"])` → `Just (Clause [Negative "q"])`.

Soluție. .

```
data Literal = Positive String | Negative String

data Clause = Clause [Literal]
```

```
4
5  instance Show Literal where
6      show (Positive s) = s
7      show (Negative s) = "tilda" ++ s
8
9  instance Show Clause where
10     show (Clause []) = ""
11     show (Clause (literal : literals)) = show literal ++
12         concatMap (("␣v␣" ++) . show) literals
13
14 complementary :: Literal -> Literal -> Bool
15 complementary (Positive s1) (Negative s2) = s1 == s2
16 complementary (Negative s1) (Positive s2) = s1 == s2
17 complementary _             _             = False
18
19 decompose :: [a] -> [(a, [a])]
20 decompose [] = []
21 decompose (h : t) = (h, t) : map (\(x, xs) -> (x, h : xs))
22                                 (decompose t)
23
24 resolve :: Clause -> Clause -> Maybe Clause
25 resolve (Clause c1) (Clause c2) = case resolvents of
26     []            -> Nothing
27     resolvent : _ -> Just resolvent
28   where
29     d1 = decompose c1
30     d2 = decompose c2
31     resolvents = [ Clause $ c1' ++ c2' | (l1, c1') <- d1,
32                                          (l2, c2') <- d2,
33                                          complementary l1 l2 ]
```

De remarcat că, datorită evaluării leneșe, elementele din *list comprehension* se generează doar până la prima pereche de la literali complementari! □

Barem.

(a)
- 1p antet `Show Literal`
- 1p `show Positive`
- 1p `show Negative`
- 1p antet `Show Clause`
- 1p `show` clauza vidă
- 5p `show` clauza cu cel puțin un literal
  - 1p parcurgere lista de literali
  - 1p `show` pe fiecare literal
  - 1p intercalare conector
  - 1p tratare particulară a primului literal
  - 1p tip corect al rezultatului (e.g. `String`, nu listă de `String`-uri)

(b)
- 1p caz de bază
- 1p aplicație recursivă pe restul listei

- 5p adăugarea primului element al listei în fața celei de-a doua componente a fiecărei perechi, cu păstrarea primei componente
- 3p adăugarea unei noi perechi, cu primul element al listei drept primă componentă a perechii

(c)
- 1p descompunere prima clauză
- 1p descompunere a doua clauză
- 1p iterare pe rezultele descompunerii primei clauze
- 1p iterare pe rezultele descompunerii celei de-a doua clauze
- 1p verificarea complementarității literalilor curenți
- 2p construcția rezolventului
- 1p diferențiere `Just` vs `Nothing`
- 1p caz `Nothing`
- 1p caz `Just`