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. Scrieți o expresie lambda care conține exact 5 apariții ale variabilei x, toate **legate** în raport cu întreaga expresie. În plus, trebuie să existe o subexpresie în raport cu care 3 dintre cele 5 apariții să fie **libere**, și să **nu** existe o subexpresie în raport cu care mai mult de 3 dintre apariții să fie libere. **Justificați!**

   Soluție.

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

   Barem.

   - 5p 5 apariții legate în raport cu întreaga expresie
   - 3p 3 apariții libere în raport cu o subexpresie
   - 1p apartenența celor 3 apariții libere la aceeași subexpresie, nu la subexpresii diferite
   - 1p absența unei expresii cu mai mult de 3 apariții libere.

2. Definiți în Racket funcția `skip-sum`, care însumează anumite elemente ale unei liste, astfel: primul număr contribuie la sumă; dacă un număr contribuie la sumă, atunci se sare peste același număr de elemente și se continuă. De exemplu, `(skip-sum '(2 3 3 1 2 4 5 2 1 2 3 1))` → `10`, unde numerele subliniate contribuie la sumă (după includerea lui `2`, se sare peste următoarele 2 elemente etc.).

   (a) Implementați funcția `skip-sum` utilizând **recursivitate explicită pe stivă**.

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

   Soluție.

   (a) Implementare:

   ```
   1 (define (skip-sum L [skip 0])
   2   (cond [(null? L) 0]
   3         [(zero? skip) (+ (car L) (skip-sum (cdr L) (car L)))]
   4         [else (skip-sum (cdr L) (sub1 skip))]))
   ```

   (b) Se utilizează 5 cadre de stivă, unul inițial, și încă 4 suplimentare, după fiecare element inclus. Se acceptă și răspunsul 4, justificat corect. □

   Barem.

   (a) 8p
   - 1p lista vidă
   - 3p includere element
   - 2p ignorare element

- 2p recursivitate pe stivă

(b) 2p, numai în prezența justificării

3. Definiți în Racket funcționala `scanr`, similară cu `foldr`, dar care întoarce lista tuturor acumulatorilor intermediari, nu numai pe cel final. De exemplu, `(foldr + 0 '(1 2 3))` → `6`, în timp ce `(scanr + 0 '(1 2 3))` → `'(6 5 3 0)`. Observați că ordinea acumulatorilor reflectă ordinea de prelucrare a elementelor listei originale.

   (a) Implementați `scanr` utilizând **exclusiv funcționale** (fără recursivitate explicită).

   (b) Implementați funcția `suffixes`, care întoarce lista sufixelor unei liste, ca **aplicație parțială** a lui `scanr`. De exemplu, `(suffixes '(1 2 3))` → `'((1 2 3) (2 3) (3) ())`.

Soluție.

(a) Implementare:

```
(define (scanr f acc L)
  (foldr (lambda (x acc1)
           (cons (f x (car acc1)) acc1))
         (list acc)
         L))
```

(b) Implementare:

```
(define suffixes (curry scanr cons '()))
```

□

Barem.

(a) 7p, doar în absența recursivității explicite
- 1p funcționala
- 4p funcția trimisă funcționalei
- 2p acumulatorul inițial

(b) 3p
- 1p aplicare parțială
- 1p funcția
- 1p acumulatorul inițial

4. Definiți în Racket funcția `diagonal`, care primește o matrice potențial infinită în ambele dimensiuni, reprezentată ca un flux de fluxuri (linii), și întoarce diagonala sa principală, reprezentată tot ca un flux. De exemplu, `(stream (stream s00 s01 ...) (stream s10 s11 ...) ...)` → `(stream s00 s11 ...)`.

Soluție.

```
(define (diagonal matrix)
  (stream-map
   stream-first
   (let go ([matrix matrix])
     (stream-cons (stream-first matrix)
                  (go (stream-map stream-rest
                                  (stream-rest matrix)))))))
```

□

Barem.

- 4p eliminarea elementelor de la începutul unei linii
- 3p eliminarea elementelor de la începutul tuturor liniilor
- 2p determinarea elementului diagonal al unei linii
- 1p determinarea elementului diagonal al tuturor liniilor

5. Sintetizați tipul expresiei Haskell `filter ($ 2)`. Considerați că literalul `2` este polimorfic.

Soluție.

```
1 filter :: (a -> Bool) -> [a] -> [a]
2 ($) :: (b -> c) -> b -> c
3 2 :: Num d => d
4 b = d
5 ($ 2) :: Num d => (d -> c) -> c
6 a -> Bool = Num d => (d -> c) -> c
7 a = Num d => d -> c
8 c = Bool
9 filter ($ 2) :: Num d => [d -> Bool] -> [d -> Bool]
```

□

Barem.

- 1p tip `filter` (linia 1)
- 1p tip `($)` (linia 2)
- 1p tip `2` (linia 3)
- 2p tip `($ 2)` (liniile 4–5)
- 3p unificare tip parametru 1 `filter` cu tip `($ 2)` (liniile 6–8)
- 2p tip final (linia 9)

6. Fie în Haskell constructorul de tip `WrappedPair`, care denotă perechi ambalate într-un constructor de tip `w`, și clasa `Wrap`, ale cărei instanțe sunt „ambalaje" pentru perechi. Scrieți o posibilă instanță a clasei, evidențiind și tipul particularizat al funcției `unwrap`.

```
1 newtype WrappedPair w
2     = WP { contents :: w (Int, Bool) }
3
4 class Wrap w where
5     unwrap :: WrappedPair w -> w Int
```

Soluție.

```
1 instance Wrap Maybe where
2     unwrap :: WrappedPair Maybe -> Maybe Int
3     unwrap (WP mPair) = fmap fst mPair
```

De remarcat că definiția funcționează pentru orice `Functor`, și poate fi generalizată în consecință. □

Barem.

- 1p alegere constructor de tip unar
- 1p antet corect al clasei, fără aplicarea constructorului de tip
- 2p tip particularizat `unwrap`
- 6p implementare corectă

7. Din punct de vedere logic, propoziția $\forall x.\exists y.P(x,y)$ **NU** implică propoziția $\exists y.\forall x.P(x,y)$ (proprietea de *necomutativitate*). 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.

   Transformăm premisa în formă clauzală:

   $$\begin{gathered}\forall x.\exists y.P(x,y)\\ \forall x.P(x,f(x))\\ \{P(x,f(x))\}\end{gathered}$$

   Apoi, transformăm negația concluziei în forma clauzală:

   $$\begin{gathered}\neg\exists y.\forall x.P(x,y)\\ \forall y.\exists x.\neg P(x,y)\\ \forall y.\neg P(g(y),y)\\ \{\neg P(g(y),y)\}\end{gathered}$$

   Pentru a rezolva cele două clauze, sunt necesare legările $x \leftarrow g(y)$ și $y \leftarrow f(x)$, care conduc la legarea ciclică $x \leftarrow g(f(x))$. □

   Barem.

   - 2p transformarea premisei în formă clauzală
   - 3p transformarea negației concluziei în forma clauzală
   - 2p cele 2 legări
   - 3p motivul eșecului

8. Definiți în Prolog predicatul `increasingSubseq(+L, ?S)`, care primește o listă `L` și leagă `S`, pe rând, la câte o subsecvență strict crescătoare de elemente nu neapărat consecutive din `L`. De exemplu, interogarea `increasingSubseq([1, 3, 2], S)` leagă `S` la `[1, 3]`, `[1, 2]`, `[1]`, `[3]`, `[2]`, `[]`, dar nu la `[1, 3, 2]` sau `[3, 2]`. **Atenție!** Trebuie construite doar subsecvențele crescătoare; nu se acceptă generarea tuturor subsecvențelor și apoi păstrarea celor crescătoare.

   Soluție.

```
incSubseq([], []).
incSubseq([H|T], [H|S]) :-
    incSubseq(T, S), (S = []; S = [X|_], H < X).
incSubseq([_|T], S) :- incSubseq(T, S).
```

□

Barem.

- 1p cazul de bază
- 6p cazul de includere a elementului curent din listă
  - 2p obținerea unei subsecvențe a restului listei
  - 1p subsecvența vidă
  - 2p subsecvența nevidă, cu verificarea inegalității
  - 1p adăugarea elementului curent din listă la subsecvența din restul listei
- 3p cazul de excludere a elementului curent din listă
  - 2p obținerea unui subsecvențe a restului listei
  - 1p neincluderea elementului curent din listă în subsecvența din restul listei

9. Definiți în Prolog predicatul `nodes(?Nodes)`, care determină lista nodurilor, **fără duplicate**, dintr-un graf neorientat descris exclusiv prin fapte `edge(X, Y)`. Pentru fiecare muchie `X-Y`, există doar una dintre afirmațiile `edge(X, Y)` sau `edge(Y, X)`. **Evitați recursivitatea explicită**, utilizând în schimb **metapredicate**, **mai puțin `bagof` sau `setof`**. De exemplu, pentru definițiile `edge(a, b)`, `edge(a, c)`, `edge(b, c)`, lista este `[a, b, c]`. ***Hint***: Predicatul `nth0(?Index, ?List, ?Elem)` leagă `Index` la indicele unei apariții a lui `Elem` în `List`.

Soluție.

```
nodes(Nodes) :-
  findall(X, (arc(X, _); arc(_, X)), Ends),
  findall(X, (nth0(I, Ends, X), \+ (nth0(J, Ends, X), J < I)), Nodes).
```

□

Barem.

- 4p determinarea listei de noduri, cu duplicate
  - 2p metapredicat
  - 2p luarea în calcul a ambilor parametri ai lui `arc`
- 6p, păstrarea aparițiilor unice, fără `setof`
  - 2p metapredicat
  - 4p determinarea aparițiilor care trebuie păstrate

Nu se acordă punctaj în prezența recursivității explicite.

10. PROBLEMA. Două persoane participă la un joc; în fiecare rundă, un jucător decide fie să colaboreze cu celălalt (`Coop`), fie să îl exploateze (`Expl`). Deciziile se iau la începutul fiecărei runde, fără a cunoaște decizia adversarului din aceeași rundă. Însă, la finalul rundei, deciziile devin cunoscute, și pot influența decizia adversarului din rundele următoare. Ne propunem să modelăm diverse strategii și să simulăm interacțiunea repetată a celor doi jucători, utilizând definițiile de mai jos:

```
data Decision = Coop | Expl
type Strategy = [Decision] -> [Decision]
```

Tipul `Decision` surprinde posibilitățile de a coopera sau exploata, iar tipul `Strategy` definește strategia unui jucător, ca o funcție pe liste potențial infinite. Parametrul funcției reprezintă lista tuturor deciziilor **adversarului** de la începutul jocului, iar rezultatul întors, lista tuturor deciziilor **jucătorului curent**. Pentru a respecta regulile jocului, poziția i din rezultat poate utiliza pentru calculul său numai pozițiile $0 \ldots i-1$ din parametru. În particular, primul element al rezultatului nu poate utiliza **niciun element** al parametrului.

(a) Definiți:

- Funcția `pureStrategy :: Decision -> Strategy`, pentru construcția unei strategii bazate pe aceeași decizie în fiecare rundă, indiferent de deciziile anterioare ale adversarului.
- Strategia `exploitEvery2nd :: Strategy`, care exploatează adversarul la fiecare a doua rundă, dar în rest cooperează.
- Strategia `titForTat :: Strategy` („ochi pentru ochi"), care începe prin a coopera, apoi reflectă în runda curentă decizia adversarului din runda anterioară.

(b) Definiți strategia `grudger :: Strategy` (ranchiunoasă), care cooperează până când adversarul exploatează, iar începând cu runda următoare doar exploatează.

(c) Definiți funcția `play :: Strategy -> Strategy -> [(Decision, Decision)]`, care primește ca parametri două strategii și întoarce lista infinită a rundelor jocului. O rundă este reprezentată prin perechea deciziilor celor doi jucători. În principiu, ar putea un jucător să trișeze și să utilizeze decizia adversarului din runda curentă (sau din rundele viitoare!), contrar regulilor jocului? Cum se comportă funcția `play` dacă numai un jucător trișează, respectiv dacă ambii trișează?

Exemple:

```
>>> take 4 $ pureStrategy Expl undefined
[Expl,Expl,Expl,Expl]

>>> take 4 $ exploitEvery2nd undefined
[Coop,Expl,Coop,Expl]

>>> take 4 $
    titForTat $ [Expl,Coop,Expl] ++ undefined
[Coop,Expl,Coop,Expl]

>>> take 4 $
    grudger $ [Coop,Expl,Coop] ++ undefined
[Coop,Coop,Expl,Expl]

>>> take 4 $ play titForTat exploitEvery2nd
[(Coop,Coop),(Coop,Expl),
 (Expl,Coop),(Coop,Expl)]

>>> take 4 $ play grudger exploitEvery2nd
[(Coop,Coop),(Coop,Expl),
 (Expl,Coop),(Expl,Expl)]
```

Soluție.

```haskell
data Decision = Coop | Expl
    deriving (Eq, Show)

type Strategy = [Decision] -> [Decision]

pureStrategy :: Decision -> Strategy
pureStrategy = const . repeat

exploitEvery2nd :: Strategy
exploitEvery2nd = const $ cycle [Coop, Expl]

titForTat :: Strategy
titForTat = (Coop :)

grudger :: Strategy
grudger = (Coop :) . (++ repeat Expl) . takeWhile (== Coop)

play :: Strategy -> Strategy -> [(Decision, Decision)]
play s1 s2 = zip decisions1 decisions2
  where
    decisions1 = s1 decisions2
    decisions2 = s2 decisions1
```

Dacă numai un jucător trișează, el poate într-adevăr utiliza deciziile din prezent sau viitor ale adversarului, contrar regulilor jocului; de exemplu, când adversarul utilizează o strategie pură, care nu consultă deciziile trișorului.

Dacă ambii jucători trișează și utilizează decizia adversarului din runda curentă, contrar regulilor jocului, funcția `play` intră în buclă infinită. □

Barem.

(a)
- 3p `pureStrategy`
- 4p `cheatEvery2nd`
- 3p `titForTat`

(b)
- 4p determinarea prefixului cooperant al adversarului
- 4p comutarea pe exploatare
- 2p cooperarea în prima rundă

(c)
- 6p `play`
  - 2p construcția primei liste de decizii pe baza celei de-a doua
  - 2p construcția celei de-a doua liste de decizii pe baza primeia
  - 2p combinarea listelor
- 2p răspunsul la prima întrebare
- 2p răspunsul la a doua întrebare

Problema este inspirată din concursul lui Axelrod[1] și din cartea *Thinking Functionally with Haskell*[2].

[1] https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/axelrod.html

[2] https://www.cs.ox.ac.uk/publications/books/functional/