1.
a)
Obs: În rezolvarea acestui subiect, puteți folosi semnul \
(backslash) în loc de λ (lambda).

În Calcul Lambda se definesc:
0 := λf.λx.x
SUCC := λn.λf.λx.(f ((n f) x))

Calculați (SUCC 0).

b)
Obs: În rezolvarea acestui subiect, puteți folosi semnul \
(backslash) în loc de λ (lambda).

În Calcul Lambda se definesc:
1 := λf.λx.(f x)
SUCC := λn.λf.λx.(f ((n f) x))

Calculați (SUCC 1).

c)
Obs: În rezolvarea acestui subiect, puteți folosi semnul \
(backslash) în loc de λ (lambda).

În Calcul Lambda se definesc:
2 := λf.λx.(f (f x))
SUCC := λn.λf.λx.(f ((n f) x))

Calculați (SUCC 2).

d)
Obs: În rezolvarea acestui subiect, puteți folosi semnul \
(backslash) în loc de λ (lambda).

În Calcul Lambda se definesc:
3 := λf.λx.(f (f (f x)))
SUCC := λn.λf.λx.(f ((n f) x))

Calculați (SUCC 3).

Soluții 1:
Barem:
- reducere λn.restul: **4p**
- reducere λf interior: **3p**

- reducere λx interior: **3p**

Primul pas trebuie efectuat separat. În caz contrar, reducerea nu se punctează.

Dacă pasul 2 este combinat cu pasul 3, se scad 2 puncte.

```
   (SUCC 0)
(λn.λf.λx.(f ((n f) x)) λf.λx.x) ->b
λf.λx.(f ((λf.λx.x f) x)) ->b
λf.λx.(f (λx.x x)) ->b
λf.λx.(f x)

   (SUCC 1)
(λn.λf.λx.(f ((n f) x)) λf.λx.(f x)) ->b
λf.λx.(f ((λf.λx.(f x) f) x)) ->b
λf.λx.(f (λx.(f x) x)) ->b
λf.λx.(f (f x))

   (SUCC 2)
(λn.λf.λx.(f ((n f) x)) λf.λx.(f (f x))) ->b
λf.λx.(f ((λf.λx.(f (f x)) f) x)) ->b
λf.λx.(f (λx.(f (f x)) x)) ->b
λf.λx.(f (f (f x)))

   (SUCC 3)
(λn.λf.λx.(f ((n f) x)) λf.λx.(f (f (f x)))) ->b
λf.λx.(f ((λf.λx.(f (f (f x))) f) x)) ->b
λf.λx.(f (λx.(f (f (f x))) x)) ->b
λf.λx.(f (f (f (f x))))
```

2.

a)

Pentru funcția f de mai jos:

```
(define (f L)
  (if (or (null? L) (null? (cdr L)))
      L
      (cons (max (car L) (cadr L))
            (f (cddr L)))))
```

i) Argumentați ce tip de recursivitate (stivă/coadă) folosește f. (1p tipul, 2p argumentația)

ii) Spuneți în cuvinte ce calculează f. (2p)

iii) Definiți funcția ff, care are același efect cu f, dar folosește celălalt tip de recursivitate. (5p)

Soluție 2a:
i) stivă - apelul recursiv nu e în poziție finală.
ii) se parcurg elementele 2 câte 2 și se reține maximul din fiecare pereche; pentru o listă cu număr impar de elemente, ultimul element este copiat ca atare în rezultat.
iii)

```
; nu se acordă puncte dacă recursivitatea nu este pe coadă
(define (ff L)
  (let loop ((L L) (acc '())) ; 1p acc
    (if (or (null? L) (null? (cdr L)))
        (reverse (append L acc))     ; 1p ordinea corectă
                                     ; 1p returul acc pe cazul de bază
        (loop (cddr L) (cons (max (car L) (cadr L)) acc)))))
                              ; 2p apelul recursiv cu actualizarea acc
```

b)
Pentru funcția f de mai jos:

```
(define (f L)
  (cond ((or (null? L) (null? (cdr L))) L)
        ((equal? (car L) (cadr L)) (f (cdr L)))
        (else (cons (car L) (f (cdr L))))))
```

i) Argumentați ce tip de recursivitate (stivă/coadă) folosește f. (1p tipul, 2p argumentația)
ii) Spuneți în cuvinte ce calculează f. (2p)
iii) Definiți funcția ff, care are același efect cu f, dar folosește celălalt tip de recursivitate. (5p)

Soluție 2b:
i) stivă - al doilea apel recursiv nu e în poziție finală.
ii) se elimină duplicatele consecutive din listă.
iii)

```
; nu se acordă puncte dacă recursivitatea nu este pe coadă
(define (ff L)
  (let loop ((L L) (acc '())) ; 1p acc
    (cond ((or (null? L) (null? (cdr L))) (reverse (append L acc)))
                                 ; 1p ordinea corectă
                                 ; 1p returul acc pe cazul de bază
          ((equal? (car L) (cadr L)) (loop (cdr L) acc))
                                 ; 1p apelul cu acc nemodificat
          (else (loop (cdr L) (cons (car L) acc))))))
                                 ; 1p apelul cu acc actualizat
```

c)

Pentru funcția f de mai jos:

```
(define (f L x)
  (cond ((null? L) x)
        ((null? (cdr L)) (+ x (car L)))
        ((equal? (car L) (cadr L)) (f (cdr L) x))
        (else (f (cdr L) (+ x (car L))))))
```

i) Argumentați ce tip de recursivitate (stivă/coadă) folosește f. (1p tipul, 2p argumentația)
ii) Spuneți în cuvinte ce calculează f. (2p)
iii) Definiți funcția ff, care are același efect cu f, dar folosește celălalt tip de recursivitate. (5p)

Soluție 2c:
i) coadă - toate apelurile recursive sunt în poziție finală.
ii) se adună elementele din listă, pentru duplicatele consecutive luându-se în calcul o singură valoare.
iii)

```
; nu se acordă puncte dacă recursivitatea nu este pe stivă
(define (ff L)
  (cond ((null? L) 0) ; 1p retur pt lista vidă
        ((null? (cdr L)) (car L)) ; 1p retur pt lista de un element
        ((equal? (car L) (cadr L)) (ff (cdr L)))

                ; 1p apelul recursiv cu ignorarea elementului curent
        (else (+ (car L) (ff (cdr L))))))
              ; 2p adunarea (car L) la apelul recursiv
```

d)
Pentru funcția f de mai jos:

```
(define (f L x)
  (cond ((null? L) x)
        ((> (car L) x) (f (cdr L) x))
        (else (+ (car L) (f (cdr L) x)))))
```

i) Argumentați ce tip de recursivitate (stivă/coadă) folosește f. (1p tipul, 2p argumentația)
ii) Spuneți în cuvinte ce calculează f. (2p)
iii) Definiți funcția ff, care are același efect cu f, dar folosește celălalt tip de recursivitate. (5p)

Soluție 2d:
i) stivă - al doilea apel recursiv nu e în poziție finală.
ii) se adună o valoare x cu toate elementele din listă mai mici sau egale cu x.

iii)

```
; nu se acordă puncte dacă recursivitatea nu este pe stivă
(define (ff L x)
  (let loop ((L L) (acc x)) ; 1p acc
    (cond ((null? L) acc)   ; 1p returul acc pe cazul de bază
          ((> (car L) x) (loop (cdr L) acc))
                         ; 1p apelul cu acc nemodificat
          (else (loop (cdr L) (+ acc (car L)))))))
                          ; 2p apelul cu acc actualizat
```

3.
a)
Atenție: Acest exercițiu nu se punctează decât dacă este rezolvat fără a folosi recursivitate
explicită (deci nici letrec sau named let). Soluțiile explicit recursive se punctează cu 0.

Definiți, în Racket, funcția odd-pos care selectează doar elementele aflate pe poziții impare într-o listă L.
Lista este indexată de la 0.
Vă poate fi de folos (dar nu este neapărat necesară) funcția range (ex: (range 5) => '(0 1 2 3 4) ; (range 8) => '(0 1 2 3 4 5 6 7)).

ex:
(odd-pos '(5 6 5 4 3 4 5)) => '(6 4 4)

Soluție 3a:

```
; cu map + filter
(define (odd-pos L)
  (let ((p (map cons L (range (length L)))))
; 4p împerechere element-poziție (2p folosire map, nu un zip inventat)
    (map car ; 2p extragerea elementelor listei din perechi
         (filter (λ(x) (odd? (cdr x))) p))))
            ; 4p filtrarea pozițiilor impare (2p predicat corect)

; cu fold
(define (odd-pos2 L)
  (reverse ; 2p ordine corectă
   (second ; 2p rezultat corect (doar elementele cerute, fără alte informații)
    (foldl (λ (x acc)
           ; 1p ordine corectă pt argumentele funcției de la fold
```

```
            (if (first acc)
          ; 2p metodă de a distinge între pozițiile pare și impare
                (list #f (cons x (second acc)))

                      ; 1p noul acc pe cazul de poziție impară
                (list #t (second acc))))
                      ; 1p noul acc pe cazul de poziție pară
           (list #f '())
                     ; 1p acc inițial
          L))))
```

b)
Atenție: Acest exercițiu nu se punctează decât dacă este rezolvat fără a folosi recursivitate
explicită (deci nici letrec sau named let). Soluțiile explicit recursive se punctează cu 0.

Definiți, în Racket, funcția alternate care primește o listă L și 2 funcții f1 și f2 și aplică alternativ f1 și f2 pe elementele lui L. Vă poate fi de folos (dar nu este neapărat necesară) funcția make-list
(ex: (make-list 4 +) => '(#<procedure:+> #<procedure:+> #<procedure:+> #<procedure:+>))

ex:
(alternate '(1 2 3 4 5 6 7) add1 sub1) => '(2 1 4 3 6 5 8)

Soluție 3b:

```
; cu map
(define (alternate L f1 f2)
  (let* ((l  (length L))
         (fL (flatten (make-list l (list f1 f2)))))
; 4p crearea listei de funcții (2p aplatizare cu flatten sau apply)
    (map (λ (f x) (f x)) (take fL l) L)))
; 4p aplicarea listei de funcții pe L
; 2p lungimea corectă a listei de funcții

; cu fold
(define (alternate2 L f1 f2)
  (reverse ; 2p ordine corectă
   (second ; 2p rezultat corect (doar elementele cerute, fără alte informații)
    (foldl (λ (x acc)
           ; 1p ordine corectă pt argumentele funcției de la fold
```

```
            (if (first acc)
          ; 2p metodă de a distinge între pozițiile pare și impare
                (list #f (cons (f1 x) (second acc)))
                      ; 1p noul acc pe cazul aplicării f1
                (list #t (cons (f2 x) (second acc)))))
                      ; 1p noul acc pe cazul aplicării f2
           (list #t '())
                     ; 1p acc inițial
          L))))
```

c)
Atenție: Acest exercițiu nu se punctează decât dacă este rezolvat fără a folosi recursivitate
explicită (deci nici letrec sau named let). Soluțiile explicit recursive se punctează cu 0.

Definiți, în Racket, funcția compare-seq care primește o listă numerică L și întoarce o listă de elemente de tip 'LT, 'EQ sau 'GT după relația de ordine între elementele consecutive ale listei L ('LT atunci când primul e mai mic decât al doilea, 'EQ când sunt egale, 'GT când primul e mai mare).

ex:
(compare-seq '(1 1 2 1 4 3)) => '(EQ LT GT LT GT) (explicație: primul EQ descrie relația dintre elementele de pe primele 2 poziții, apoi LT pentru elementele de pe pozițiile 2 și 3, apoi GT pentru elementele de pe pozițiile 3 și 4, etc.)

Soluție 3c:

```
(define (compare-seq L)
  (let* ((l  (length L))
         (L1 (take L (- l 1)))
                ; 2p crearea listei fără ultimul element
         (L2 (cdr L)))
                ; 2p crearea listei fără primul element
                ; 4p aplicarea funcției binare pe cele 2 liste
                ; 2p funcția binară a lui map
    (map (λ (x y) (cond ((= x y) 'EQ)
                        ((< x y) 'LT)
                        ((> x y) 'GT))) L1 L2)))
```

d)
Atenție: Acest exercițiu nu se punctează decât dacă este rezolvat fără a folosi recursivitate

explicită (deci nici letrec sau named let). Soluțiile explicit recursive se punctează cu 0.

Definiți, în Racket, funcția eq-sums? care primește o listă numerică L cu număr par de elemente și întoarce true dacă toate sumele provenite din adunarea elementelor de pe poziții simetrice față de mijlocul listei sunt egale, respectiv false în caz contrar.

ex:
(eq-sums? '(1 4 2 5 3 6)) => #t (explicație: 1+6 = 4+3 = 2+5)
(eq-sums? '(1 4 5 1 3 6)) => #f (explicație: 1+6 = 4+3 != 5+1)

Soluție 3d:
```
(define (eq-sums? L)
  (or (null? L) ; 0p cazul de listă vidă
      (let ((R (reverse L))) ; 2p inversarea listei
        (null? (filter (λ (x) (not (equal? x (+ (car L) (car R)))))
                ; 2p predicat corect filter
                ; 2p prelucrare filter (aici null?)
                       (map + L R))))))
               ; 4p adunarea celor 2 liste
```

Subiectul 4
==========

Varianta 1

Pornind de la fluxul numerelor naturale din Racket, definiți fluxul fluxurilor de numere naturale consecutive, din 2 în 2, din 3 în 3 ș.a.m.d. Exemplu: ((0 1 2 3 ...) (0 2 4 6 ...) (0 3 6 9 ...) ...). Considerați că naturals este deja definit și că puteți utiliza orice funcționale pe fluxuri, discutate la curs și laborator.

Hint: În loc să construiți de la zero fiecare flux interior, gândiți-vă cum îl puteți obține pe baza fluxului anterior.

Rezolvare:

```
(define multiples
  (stream-cons naturals
               (stream-map (λ (s) (stream-zip-with + s naturals))
                           multiples)))
```

Barem:

5p: mecanismul de generare a întregului flux (explicit sau implicit)
5p: definirea elementului nou pe baza celui vechi

Varianta 2

Definiți în Racket următorul flux infinit: ((1) (1 2 1) (1 2 3 2 1) (1 2 3 4 3 2 1) ...).

Hint: În loc să construiți de la zero fiecare listă interioară, gândiți-vă cum o puteți obține pe baza listei anterioare.

Rezolvare:

```
(define pyramids
  (stream-cons '(1)
               (stream-map (λ (L)
                             (append '(1) (map add1 L) '(1)))
                           pyramids)))
```

Barem:

5p: mecanismul de generare a întregului flux (explicit sau implicit)
5p: definirea elementului nou pe baza celui vechi

Varianta 3

Pornind de la un flux infinit de liste, construiți fluxul reuniunilor parțiale ale acestuia, în care lista de pe poziția i conține reuniunea primelor i liste din fluxul inițial, iar prima listă este vidă. Atenție! Reuniunile NU trebuie să conțină duplicate! Exemplu: pentru fluxul ((2 3) (3 4 5) (2 4) (1) ...) se obține fluxul (() (2 3) (2 3 4 5) (2 3 4 5) (2 3 4 5 1) ...).

Hint: Exploatați faptul că fluxurile de față sunt infinite, pentru a prelucra simultan fluxul rezultat cu cel inițial.

Rezolvare:

```
(define (partial-unions sets)
  (letrec
      ([out
        (stream-cons
         '()
         (stream-zip-with (λ (union set)
                            (append union
                                    (filter (λ (x)
                                              (not (member x union)))
                                            set)))
                          out
                          sets))])
    out))
```

Barem:

5p: mecanismul de generare a întregului flux (explicit sau implicit)
5p: definirea elementului nou pe baza celui vechi, din care:
  2p: reuniunea
  3p: absența duplicatelor

Varianta 4

Fie programul Haskell de mai jos:

```
lst = [(1+2, 3+4), (3+4, 5+6), (5+6, 7+8)]

find ((7,x) : _) = x
find (_ : pairs) = find pairs
```

Descrieți cum decurge, pas cu pas, evaluarea expresiei (find lst).

Rezolvare:

```
find lst
find ((1+2, 3+4) : ...)
find ((3, 3+4) : ...)
find ((3+4, 5+6) : ...)
find ((7, 5+6) : ...)
5+6
11
```

Barem:

2p: întâi se evaluează doar 1+2 la 3
2p: apoi se evaluează doar 3+4 la 7
2p: se extrage 5+6
2p: se evaluează 5+6 la 11
2p: nu se evaluează altceva

Subiectul 5

Varianta 1

Sintetizați, pas cu pas, tipul următoarei funcții în Haskell:

```
f x y = head x y
```

Atenție! Simpla menționare a tipului final, fără pași intermediari, nu va fi punctată!

Rezolvare:

```
f :: a -> b -> c
x :: a
y :: b
head :: [d] -> d
a = [d]
head x = d
d = e -> g
e = b
c = g
f :: [b -> g] -> b -> g
```

Barem

0p pentru simpla scriere a tipului final, fără niciun pas intermediar.
Depunctări dacă nu rezultă cum s-a ajuns de la un pas la altul.

Varianta 2

Sintetizați, pas cu pas, tipul următoarei funcții în Haskell:

```
f x y = fst x : map (snd x) y
```

Atenție! Simpla menționare a tipului final, fără pași intermediari, nu va fi punctată!

Rezolvare:

```
f :: a -> b -> c
x :: a
y :: b
fst :: (d, e) -> d
snd :: (d, e) -> e
a = (d, e)
fst x :: d
snd x :: e
map :: (g -> h) -> [g] -> [h]
e = g -> h
b = [g]
c = [h]
d = h
f :: (h, g -> h) -> [g] -> [h]
```

Notă: Am folosit aceleași variable de tip la fst și snd pentru concizie, întrucât se aplică asupra aceluiași parametru, x. În mod normal, se utilizează alte variabile de tip, e.g. `snd :: (g, h) -> h`.

Barem

0p pentru simpla scriere a tipului final, fără niciun pas intermediar.
Depunctări dacă nu rezultă cum s-a ajuns de la un pas la altul.

Varianta 3

Sintetizați, pas cu pas, tipul următoarei funcții în Haskell:

```
f x y = [tail x ++ fst y, snd y]
```

Atenție! Simpla menționare a tipului final, fără pași intermediari, nu va fi punctată!

Rezolvare:

```
f :: a -> b -> c
x :: a
y :: b
tail :: [d] -> [d]
a = [d]
tail x :: [d]
fst :: (e, g) -> e
snd :: (e, g) -> g
b = (e, g)
fst y :: e
snd y :: g
e = [d]
g = [d]  (din aceeași listă cu elemente de tipul e)
f :: [d] -> ([d], [d]) -> [[d]]
```

Notă: Am folosit aceleași variable de tip la fst și snd pentru concizie, întrucât se aplică asupra aceluiași parametru, y. În mod normal, se utilizează alte variabile de tip, e.g. snd :: (h, i) -> i.

Barem

0p pentru simpla scriere a tipului final, fără niciun pas intermediar.
Depunctări dacă nu rezultă cum s-a ajuns de la un pas la altul.

Varianta 4

Sintetizați, pas cu pas, tipul următoarei funcții în Haskell:

```
f x y = (f x, f y)
```

Atenție! Simpla mențiune a tipului final, fără pași intermediari, nu va fi punctată!

Rezolvare:

```
f :: a -> b -> c
x :: a
y :: b
a = b
f x :: a -> c
```

```
f y :: a -> c
c = (a -> c, a -> c)
```
Eroare de sinteză, întrucât expresia de tip din dreapta conține strict variabila c.

Barem

0p pentru simpla menţiune a erorii, fără niciun pas intermediar. Depunctări dacă nu rezultă cum s-a ajuns de la un pas la altul.

Subiectul 6

Varianta 1

Supraîncărcaţi în Haskell reprezentarea sub formă de şir de caractere pentru funcţii unare, cu parametru numeric, astfel încât o astfel de funcţie să fie vizualizată printr-o listă de perechi, care conţin pe prima poziţie numerele de la 0 la 5, iar pe a doua, valorile funcţiei în aceste puncte. De exemplu, show (*2) = "[(0,0),(1,2),(2,4),(3,6),(4,8),(5,10)]".

Atenţie la constrângerile necesare la instanţierea clasei!

Rezolvare:

```
instance (Num a, Show a, Enum a, Show b) => Show (a -> b) where
    show f = show $ map (\x -> (x, f x)) [0..5]
```

Barem:

1p Show (a -> b)
1p Num a
1p Show a
1p Show b
6p implementare show
  4p valorile funcţiei
  2p punctele efective

Nu se depunctează absenţa constrângerii (Enum a).

Varianta 2

Supraîncărcați în Haskell operatorul de ordonare a funcțiilor unare, cu parametru numeric, astfel încât funcția f este mai mică sau egală cu funcția g dacă valorile lui f în punctele 0..100 sunt mai mici sau egale cu valorile lui g în aceleași puncte. Comparațiile se fac pentru aceiași parametri. De exemplu, (+1) <= (+2), întrucât (+1) 0 <= (+2) 0, (+1) 1 <= (+2) 1 etc.

Atenție la constrângerile necesare la instanțierea clasei!

Rezolvare:

```
instance (Num a, Enum a, Ord b) => Ord (a -> b) where
    f <= g = and $ zipWith (<=) (map f [0..100]) (map g [0..100])
```

Barem:

```
1p Ord (a -> b)
1p Num a
1p Ord b
7p implementare (<=)
  2p aplicare f pe cele 100 de puncte
  2p aplicare g pe cele 100 de puncte
  3p comparare valoare cu valoare
```

Nu se depunctează absența constrângerii (Enum a).

Varianta 3

Scrieți în Haskell o posibilă instanță a clasei de mai jos, conținând o implementare neconstantă a funcției convert.

```
class Convertor c where
    convert :: Num a => [a] -> c a
```

Rezolvare:

```
instance Convertor Maybe where
    convert [] = Nothing
    convert xs = Just $ sum xs
```

Barem:

0p pentru implementare constantă a lui convert.
Nu e obligatorie instanțierea cu Maybe; se poate de exemplu și cu [].

Varianta 4

Scrieți în Haskell o posibilă instanță a clasei de mai jos, conținând o implementare neconstantă a funcției merge.

```
class Merger m where
    merge :: m a -> m a -> m [a]
```

Rezolvare:

```
instance Merger Maybe where
    merge (Just x) (Just y) = Just [x, y]
    merge _        _        = Nothing
```

Barem:

0p pentru implementare constantă a lui convert.
Nu e obligatorie instanțierea cu Maybe; se poate de exemplu și cu [].

Subiectul 7
==========

Varianta A

Avem premisele, date în formă clauzală:

```
{ ¬P(x, y, z), ¬Q(z, x), R(x, y) }
{ P(Ion, Test, George) }
{ Q(George, Ion) }
```

Demonstrați *folosind rezoluția* că este adevărat R(Ion, Test).

Soluție:

Sunt 3 pași de rezoluție, cu P, Q, și R din prima clauză.

De exemplu:

```
¬R(Ion, Test)          // concluzia negată
¬P(x, y, z) V ¬Q(z, x) V R(x, y)
------------------------------ x <- Ion, y <- Test
¬P(Ion, Test, z) V ¬Q(z, Ion)
Q(George, Ion)
------------------------------ z <- George
¬P(Ion, Test, George)
P(Ion, Test, George)
------------------------
clauza vidă
```

Barem:

+3p per pas de rezoluție, -1p la fiecare pas dacă lipsește substituția sau este incorectă
+1p pentru imaginea de ansamblu și finalizarea corectă

Varianta B

Avem premisele, date în formă clauzală:

```
{ ¬P(x, y), ¬Q(x, y) }
{ ¬R(x, y), Q(x, y) }
{ P(x, y), R(x, y) }
{ P(Armand, Cadou) }
{ ¬P(Claudia, Cadou) }
```

Demonstrați *folosind rezoluția* că este adevărat Q(Claudia, Cadou).

Soluție:

Sunt 3 pași necesari, pentru
¬Q(Claudia, Cadou)
¬R(x, y), Q(x, y)

P(x, y), R(x, y)
¬P(Claudia, Cadou)

De exemplu:

¬Q(Claudia, Cadou)   // concluzia negată
¬R(x, y) V Q(x, y)
------------------ x <- Claudia, y <- Cadou
¬R(Claudia, Cadou)
P(x, y) V R(x, y)
------------------ x <- Claudia, y <- Cadou
P(Claudia, Cadou)
¬P(Claudia, Cadou)
----
clauza vidă

Barem:

+3p per pas de rezoluție, -1p la fiecare pas dacă lipsește substituția sau este incorectă
+1p pentru imaginea de ansamblu și finalizarea corectă

Varianta C

Avem premisele, date în formă clauzală:

{ ¬P(x, y), ¬Q(z, t, x), R(t, z, y) }
{ Q(Ionuț, Aurel, Carioca) }
{ P(Carioca, Pachet) }

Demonstrați *folosind rezoluția* că este adevărat R(Aurel, Ionuț, Pachet).

Soluție:

Sunt 3 pași de rezoluție, cu P, Q, și R din prima clauză.

De exemplu:

¬P(x, y) V ¬Q(z, t, x) V R(t, z, y)
¬R(Aurel, Ionuț, Pachet) // concluzia negată
---------- t <- Aurel, z <- Ionuț, y <- Pachet
¬P(x, Pachet) V ¬Q(Ionuț, Aurel, x)
Q(Ionuț, Aurel, Carioca)
---------- x <- Carioca
¬P(Carioca, Pachet)
P(Carioca, Pachet)
---
clauza vidă

Barem:

+3p per pas de rezoluție, -1p la fiecare pas dacă lipsește substituția sau este incorectă
+1p pentru imaginea de ansamblu și finalizarea corectă

Varianta D

Avem premisele, date în formă clauzală:

{ ¬P(x), ¬Q(y, x), ¬T(z), ¬R(z, x) }
{ P(Cadou) }
{ Q(12, Cadou) }
{ T(21) }

Demonstrați *folosind rezoluția* că este adevărat ¬R(21, Cadou).

Soluție:

Sunt 4 pași de rezoluție, cu P, Q, T și R din prima clauză.

De exemplu:

¬P(x) V ¬Q(y, x) V ¬T(z) V ¬R(z, x)
Q(12, Cadou)
---------- x <- Cadou, y <- 12
¬P(Cadou) V ¬T(z) V ¬R(z, Cadou)
P(Cadou)

¬T(z) V ¬R(z, Cadou)
T(21)
---------- z <- 21
¬R(12, Cadou)
R(21, Cadou)   // concluzia negată

Clauza vidă

Barem:

+2p per pas de rezoluție, -1p la fiecare pas dacă lipsește substituția sau este incorectă
+2p pentru imaginea de ansamblu și finalizarea corectă

Subiectul 8
==========

Se respectă forma răspunsului 1p
Caz de bază 1p
Ordinea raspunsurilor 2p
Corectitudinea 6p
Mai multe soluții -3p
Cut neexplicat sau plasat nediscriminatoriu -2p
Singletons -1p

Varianta A

Implementați în Prolog predicatul identify(+ListsIn, -ListsOut) care pentru o listă de liste numerice, produce o listă de perechi, fiecare pereche (N, L) corespunzând unei liste L din ListsIn, iar numărul N fiind:
- maximul listei, dacă L este sortată crescător;
- primul element din L, altfel.

Notă: listele din ListsIn nu conțin duplicate.
Hint: puteți folosi predicatul predefinit sort/2, sort(+List, -Sorted).

De exemplu, interogarea identify([[1,2,3], [2,1,3], [4,5,6], [7,6,9]], X).
este adevărată pentru X = [(3, [1, 2, 3]), (2, [2, 1, 3]), (6, [4, 5, 6]), (7, [7, 6, 9])].

Puteți folosi predicate ajutătoare. Predicatul trebuie să întoarcă cel mult o soluție. Dacă ați folosit predicatul cut, explicați plasarea acestuia. Nu lăsați variabile singleton.

Soluție:

```
identify([], []).
identify([L|LL], [(H, L) | LLOut]) :-
     sort(L, L), !, reverse(L, [H|_]), identify(LL, LLOut).
identify([[H|T]|LL], [(H,[H|T])|LLOut]) :- identify(LL, LLOut).
```

Varianta B

Implementați în Prolog predicatul noRedundancies(+ListsIn, -ListsOut). ListsIn este o listă de liste. ListsOut este o sublistă din ListsIn. Ne interesează să păstrăm la ieșire doar liste care conțin cel puțin un element nou față de listele mai din stânga.

Hint: puteți folosi predicatul predefinit subset/2, subset(+SubSet, +Set).

De exemplu, interogarea noRedundancies([[3,1], [2,5,6,7], [4,5,6], [1,2,3]], X). va face legarea:
X = [[3, 1], [2, 5, 6, 7], [4, 5, 6]].
pentru că toate elementele din lista [1,2,3] sunt deja conținute în liste mai din stânga.

Iar interogarea noRedundancies([[5,6,7,3], [4,5], [4,5,6], [1,2], [1,3,2]], X).
va face legarea:
X = [[5, 6, 7, 3], [4, 5], [1, 2]].
pentru că nici [1,2,3], nici [4,5,6] nu conțin elemente noi față de listele mai din stânga lor.

Puteți folosi predicate ajutătoare. Predicatul trebuie să întoarcă cel mult o soluție. Dacă ați folosit predicatul cut, explicați plasarea acestuia. Nu lăsați variabile singleton.

Soluție:

```
aux3b([], [], _).
aux3b([L|LL], LLOut, Elements) :-
     subset(L, Elements), !,
     aux3b(LL, LLOut, Elements).
aux3b([L|LL], [L|LLOut], Elements) :-
     append(Elements, L, Elements1),
     aux3b(LL, LLOut, Elements1).
noRedundanciesB(LL, LLOut) :- aux3b(LL, LLOut, []).
```

Varianta C

Implementați în Prolog predicatul subDown(+List, -Sub) care leagă Sub la o sublistă descrescătoare a lui List (care este nevidă), conținând următoarele elemente:
- primul E1 element din List
- următorul element Ei mai mic decât E1
- următorul element Ej mai mic decât Ei
- și așa mai departe.

De exemplu, sunt adevărate:
```
subDown([5,1,2,3,0,4,3, 6], X).   cu X = [5, 1, 0]
subDown([1,2,0,3,4,3], X).   cu X = [1, 0]
subDown([1,2,3,4], X).   cu X = [1]
```

Puteți folosi predicate ajutătoare. Predicatul trebuie să întoarcă cel mult o soluție. Dacă ați folosit predicatul cut, explicați plasarea acestuia. Nu lăsați variabile singleton.

Soluție:

```
aux2([], _, []).
aux2([H|T], C, [H|ST]) :- H < C, !, aux2(T, H, ST).
aux2([_|T], C, ST) :- aux2(T, C, ST).
subDown([H|T], [H|ST]) :- aux2(T, H, ST).
```

Varianta D

Implementați în Prolog predicatul subUp(+List, -Sub) care leagă Sub la o sublistă crescătoare a lui List (care este nevidă), conținând următoarele elemente:
- primul E1 element din List
- următorul element Ei mai mare decât E1
- următorul element Ej mai mare decât Ei
- și așa mai departe.

De exemplu, sunt adevărate:
subUp([1,2,0,3,4,3], X). cu X = [1, 2, 3, 4]
subUp([5,1,2,3,4,3, 6], X). cu X = [5, 6]
subUp([5,1,2,3,4,3], X). cu X = [5]

Puteți folosi predicate ajutătoare. Predicatul trebuie să întoarcă cel mult o soluție. Dacă ați folosit predicatul cut, explicați plasarea acestuia. Nu lăsați variabile singleton.

Soluție:

```
aux([], _, []).
aux([H|T], C, [H|ST]) :- H > C, !, aux(T, H, ST).
aux([_|T], C, ST) :- aux(T, C, ST).
subUp([H|T], [H|ST]) :- aux(T, H, ST).
```

Subiectul 9
==========

Varianta A

Implementați folosind metapredicate (forall, findall, bagof, setof), și fără recursivitate explicită, predicatul counts(+List, -ListPlus), care primește în List o listă de liste de elemente oarecare, și leagă ListPlus o listă de liste, fiecare element LP din ListPlus fiind bazat pe o listă L din List: primul element este numărul elementelor *diferite* din L, iar restul elementelor sunt elementele din L.

Exemplu:

counts([[a], [1, 2, 3, 4, 3], [x, y, z], [d, 1, d, 1], [e, 1,1,1,1,1,1]], X) leagă X la [[1,a],[4,1,2,3,4,3],[3,x,y,z],[2,d,1,d,1],[2,e,1,1,1,1,1,1]].

Sol:

```
counts(List, ListPlus) :-
     findall([Len|L],
          (
               member(L, List),
               setof(E, member(E, L), Es),
               length(Es, Len)
          ),
          ListPlus).
```

Barem:
Numărare elemente unice: 3p
Aplicare mecanism numărare elemente unice pe fiecare listă: 4p
Asamblare corectă: 3p

Varianta B

Implementați folosind metapredicate (forall, findall, bagof, setof), și fără recursivitate explicită, predicatul heads(+List, +Limit, -Heads) care primește o listă de liste, fiecare listă fiind formată dintr-un prim element și restul listei (restul poate fi și vid). Predicatul leagă Heads la lista acelor prime elemente în ale căror resturi de liste există cel puțin Limit elemente *diferite*.

Exemplu:

Pentru lista List = [[a], [b, 1, 2, 3, 4, 3], [c, x, y, x, y], [d, 1], [e, 1,1,1,1,1,1,1,1]]

heads(List, 2, X) leagă X la [b, c] pentru că doar listele cu b și cu c au cel puțin 2 elemente diferite, iar
heads(List, 4, X) leagă X la [b] pentru că în lista lui c sunt doar 2 elemente diferite.

Sol:

```
heads(List, Limit, Heads) :-
     findall(X,
          (
               member([X|R], List),
               setof(E, member(E, R), L),
               length(L, Len), Len >= Limit
          ),
          Heads).
```

Barem:
Găsire elemente unice: 4p
Verificarea numărului de elemente unice: 2p
Asamblarea soluției: 3p

Varianta C

Implementați folosind metapredicate (forall, findall, bagof, setof), și fără recursivitate explicită, predicatul index(+List, -Index) care primește o listă de elemente și întoarce o listă de perechi, fiecare pereche fiind formată dintr-un element E din List și numărul de apariții ale lui E în List. Hint: în Index elementele apar sortate.

Exemplu:
index([8, 1, 2, 3, 1, 2, 5, 4, 7, 6, 2, 5], X) leagă X la [(1, 2), (2, 3), (3, 1), (4, 1), (5, 2), (6, 1), (7, 1), (8, 1)]

Sol:

```
index(List, Index) :-
     setof((E, Times),
          L^(
          member(E, List),
          findall(X, (member(X, List), E = X), L),
          length(L, Times)
          ),
          Index).
```

sau

```
index(List, Index) :-
     setof(E, member(E, List), Es),
     findall((E, Times),
          (   member(E, Es),
               findall(X, (member(X, List), E = X), L),
               length(L, Times)
          ),
          Index).
```

Barem:
Găsirea elementelor fără duplicate: 4p
Numărul de apariţii pentru fiecare element: 2p
Asamblarea soluţiei: 2p

Varianta D

Implementaţi folosind metapredicate (forall, findall, bagof, setof), şi fără recursivitate explicită, predicatul singles(+List, -Singles), care filtrează în Singles elementele din List care apar o singură dată. Rezultatul este ordonat.

Exemplu:

singles([8,1,2,3,1,2,5,4,7,6,2,5], X) leagă X la [3, 4, 6, 7, 8].

Sol:

```
singles(List, Singles) :-
     setof(E,
```

```
            L^(   member(E, List),
                  findall(X, (member(X, List), E = X), L),
                  length(L, 1)
            ),
            Singles).
```

Barem:
Verificarea dacă un element este unic: 4p
Verificarea unicității pentru toate elementele: 4p
Asamblarea soluției: 2p