Examen PP/2CB/14.06.23 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

Numele și grupa	1	2	3	4	5	6	7	8	9	10

1. În Calculul Lambda se definesc:

```
add = λm.λn.λf.λx.((m f) ((n f) x))
two = λf.λx.(f (f x))
```

Efectuați primele 3 beta-reduceri din calculul ((add two) two), folosind reducere stânga-dreapta (puteți păstra notația `two` acolo unde nu este necesar să explicitați acest termen).

Soluție:

```
((add two) two) =
((λm.λn.λf.λx.((m f) ((n f) x)) two) two) ->
(    λn.λf.λx.((two f) ((n f) x))    two) ->
       λf.λx.((two f) ((two f) x))       =
λf.λx.((λf.λx.(f (f x)) f) ((two f) x))  ->
λf.λx.(λx.(f (f x)) ((two f) x))         (până aici era suficient)
                                          ->
λf.λx.(f (f ((two f) x)))                =
λf.λx.(f (f ((λf.λx.(f (f x)) f) x)))    ->
λf.λx.(f (f (λx.(f (f x)) x)))           ->
λf.λx.(f (f (f (f x))))
```

Barem: 3p, 3p, 4p (în această ordine, pentru cele 3 beta-reduceri)
Punctajul se înjumătățește dacă reducerea e corectă dar nu este stânga-dreapta.

2. Știind că or din Racket este o funcție nestrictă care (pe cazul de or binar) funcționează după următoarele axiome:

```
or(true, y) = true
or(false, y) = y,
```

argumentați pe scurt ce fel de recursivitate (pe stivă/pe coadă) vor genera apelurile (f '(2 3)), respectiv (g '(2 3)). **(8p)**
De asemenea, precizați la ce se evaluează aceste apeluri. **(2p)**

```
(define (f L)
  (or (f (map add1 (reverse L)))
      (odd? (car L))))
```

```
(define (g L)
  (or (odd? (car L))
      (g (map add1 (reverse L)))))
```

Soluție:
f și g generează aceleași noi apeluri recursive care rezultă într-o rulare la infinit:
(f '(2 3)) -> (f '(4 3)) -> (f '(4 5)) -> (f '(6 5)) … (se tot generează noi apeluri, fără a testa paritatea primului element)
(g '(2 3)) -> (g '(4 3)) -> (g '(4 5)) -> (g '(6 5)) … (primul element va fi mereu par, rezultând în noi apeluri)

- g testează întâi paritatea primului element, iar când rezultatul testului este false, conform celei de-a doua axiome, rezultatul or-ului este înlocuit cu rezultatul apelului (g (**map** add1 (**reverse** L))), rezultând o recursivitate pe coadă (rezultatul apelului recursiv este rezultatul final)
- f realizează întâi apelul recursiv, al cărui rezultat este așteptat de funcția or pentru a decide ce face mai departe, ceea ce generează o recursivitate pe stivă

Barem:
2p (1+1) – faptul că apelurile rulează la infinit
2p – f are recursivitate pe stivă
2p – argumentație: este pe stivă pentru că rezultatul este așteptat de funcția or
2p – g are recursivitate pe coadă
2p – argumentație: este pe coadă pentru că rezultatul apelului părinte este înlocuit cu rezultatul apelului recursiv, nefiind nevoie de operații la revenirea din recursivitate

3. a) În Racket, **fără a folosi recursivitate explicită**, implementați funcția **insert-everywhere** care primește un element x și o listă L și îl inserează pe x în toate pozițiile în lista L, întorcând lista rezultatelor. **(6p)**
Ex: pentru x=3 și L = (1 2), rezultatul este ((3 1 2) (1 3 2) (1 2 3))
Funcții utile: (take L nr) – întoarce primele nr elemente din L
(drop L nr) – întoarce L fără primele nr elemente

Examen PP/2CB/14.06.23 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

(range nr) – întoarce lista (0 .. nr-1) (ex: (range 5) întoarce (0 1 2 3 4))

b) Tot **fără a folosi recursivitate explicită** și considerând implementată funcția de la punctul a, implementați funcția **insert-in-all-lists** care primește un element x și o listă de liste LL și îl inserează pe x în toate pozițiile în fiecare listă din LL, întorcând lista rezultatelor. **(4p)**

Ex: pentru x=3 și LL = ((2 1) (1 2)), rezultatul este ((3 2 1) (2 3 1) (2 1 3) (3 1 2) (1 3 2) (1 2 3))

Soluție:

```
(define (insert-everywhere x L)
  (map (λ (n) (append (take L n) (list x) (drop L n)))
       (range (+ (length L) 1))))

(define (insert-in-all-lists x LL)
  (apply append (map (λ (L) (insert-everywhere x L)) LL)))
```

Barem:

a) 2p – generare range (-1p dacă nu se generează toate cele n+1 poziții)
2p – append (-1p dacă nu se folosește append sau cons cu argumente de tip corect: element/listă)
2p – map sau fold care trece prin tot range-ul
b) 2p – map sau fold care trece prin toate listele (-1p dacă nu se aplică corect insert-everywhere pe fiecare)
2p – concatenarea listelor rezultate (cu fold sau apply)

4. Folosind funcția **insert-in-all-lists** (puteți presupune că ea există, chiar dacă nu ați implementat-o), implementați în Racket fluxul infinit (((1)) ((2 1) (1 2)) ((3 2 1) (2 3 1) (2 1 3) (3 1 2) (1 3 2) (1 2 3)) …), în care elementul de pe poziția i reprezintă toate permutările listei (1 .. i), fără să conteze ordinea în care apar aceste permutări.

Soluție:

```
(define perm-stream
  (let loop ((seed '((1))) (next 2))
    (stream-cons seed (loop (insert-in-all-lists next seed) (add1 next)))))
```

Barem:

2p – precizare primul element
2p – precizare element(e) care trebuie inserat(e)
2p – insert-in-all-lists între cele două
2p – iterație corectă pentru a genera toate elementele (named let, stream-zip-with, etc.)
2p – folosire corectă interfață pentru fluxuri (stream-cons, etc.)

5. Cunoscând div, mod :: Integral a => a -> a -> a (și apartenența la clasa Integral este singura constrângere legată de operațiile cu tipuri numerice), sintetizați tipul lui lst (nu sunt necesare explicații formale, dar sunt necesare explicații).

```
lst lim = [ (n, if null ds then Nothing else Just ds ) |
            n <- [2 .. lim], let ds = [ d | d <- [2 .. div n 2], mod n d == 0 ] ]
```

Soluție:

Obs: lst primește un număr lim și generează lista de perechi (n, divizori_n_în_afară_de_1_și_n) pentru toate numerele din intervalul [2 .. lim]. Pentru că anumite numere sunt prime și nu au divizori în afară de 1 și numărul însuși, se folosește tipul Maybe.

Din div, mod :: Integral a => a -> a -> a, rezultă: n, d :: Integral a => a (pentru că sunt argumente ale funcțiilor div, mod)
Din n <- [2 .. lim] și n, d :: Integral a => a, rezultă: lim :: Integral a => a (pentru că lim e de același tip cu n)
ds fiind o listă de d-uri, rezultă: ds :: Integral a => [a]
Din (n, if ...), rezultă că fiecare element din lista rezultat este o pereche
Din if null ds then Nothing else Just ds, rezultă că al doilea element din fiecare pereche e de tipul Integral a => Maybe [a] (știind de mai sus că ds :: Integral a => [a])
Rezultă: lst :: Integral a => a -> [(a, Maybe [a])]

Barem:

2p – constrângerea Integral a
1p – lim :: a
1p – rezultatul final e o listă

Examen PP/2CB/14.06.23 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

1p – fiecare element din listă e o pereche
1p – primul element din pereche e de tip a
2p – al doilea element din pereche e un Maybe [a] (1p Maybe, 1p [a])
2p – tipul final lst :: Integral a => a -> [(a, Maybe [a])] (asamblarea componentelor de mai sus)

6. Fie tipul Tournament care descrie un turneu de tenis prin nume și lista de puncte pe care turneul le acordă pentru victoria în diverse runde (ex: T "Roma" [20, 25, 45, 90, 180, 240, 400] înseamnă că la turneul de la Roma, cine câștigă în prima rundă primește 20 de puncte, la care se adaugă 25 pentru victorie în runda 2, 45 pentru victorie în runda 3, etc.):

```
type Name = String
type Points = Int
data Tournament = T Name [Points] deriving Show
```

Adăugați tipul Tournament la clasa Eq, știind că 2 turnee sunt egale dacă oferă același număr total de puncte învingătorului turneului și aceste puncte se obțin după câștigarea aceluiași număr de meciuri.

Soluție:

```
instance Eq Tournament where
  T _ points1 == T _ points2 = sum points1 == sum points2 && length points1 == length points2
```

Barem:
2p – sintaxă instance (1p instance Eq … where, 1p folosirea tipului Tournament)
2p – definirea comportamentului funcției == (sau /=)
2p – pattern match pentru a accesa points
2p – condiția ca suma punctelor să fie egală
2p – condiția ca lungimea turneelor să fie egală

7. Traduceți în FOL (first order logic) propoziția: „Există un animal mai mic decât toți câinii cărora le place să înoate".

Soluție:
$\exists X \bullet (animal(X) \wedge \forall Y \bullet ((câine(Y) \wedge place\_să\_înoate(Y)) \Rightarrow mai\_mic(X, Y)))$

Barem:
1p – animal(X) cuantificat existențial
2p – folosire ∧ la ceea ce am cuantificat existențial (nu ⇒ sau altă conectivă)
1p – câine(Y) cuantificat universal
1p - folosire ∧ pentru condiția de a îi plăcea să înoate
2p – folosire ⇒ la ceea ce am cuantificat universal
1p – predicat care surprinde relația mai_mic(X, Y)
2p – asamblare corectă a tuturor componentelor

8. Implementați, în Prolog, predicatele **prefix**(+A, +B) **(5p)** și **sufix**(+A, +B) **(5p)**, care primesc 2 liste **nevide** A și B și verifică dacă A este un prefix, respectiv sufix al listei B. (ex: prefix([1,2], [1,3,2,5]) va fi false, sufix([2,4], [0,2,4]) va fi true)

Soluție:

```
prefix(A, B) :- A = [_|_], append(A, _, B), !.
sufix(A, B) :- A = [_|_], append(_, A, B), !.
```

Soluție alternativă:

```
prefix([X], [X|_]) :- !.
prefix([X|A], [X|B]) :- prefix(A, B).

sufix(A, B) :- reverse(A, RA), reverse(B, RB), prefix(RA, RB).
```

Barem (identic pentru ambele predicate):
1p – condiția de listă nevidă (mergem până la lista de un element, sau pattern match)
4p – condiția ca toate elementele din A să fie, în această ordine, la începutul/sfârșitul lui B
(4p append sau 2p pattern match + 2p "apel" recursiv)
În varianta cu reverse: 3p inversările, 2p utilizarea corectă a predicatului prefix

Examen PP/2CB/14.06.23 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

9. Considerând implementate predicatele prefix și sufix de mai sus, definiți într-**o singură regulă** predicatul **can_decompose**(+A, +B) care primește 2 liste **nevide** A și B și verifică dacă A poate fi scris ca o concatenare de liste **nevide** X și Y astfel încât X este un prefix și Y este un sufix pentru B. Corpul regulii can_decompose trebuie să constea într-**un singur metapredicat** (satisfacerea acestuia generează răspunsul true, nesatisfacerea sa generează false).
Ex: can_decompose([1,1,3],[1,3]) este true, pentru că [1] este prefix pentru [1,3], iar [1,3] este sufix pentru [1,3]

Soluție:

```
can_decompose(A, B) :- findall(1, (append(X, Y, A), prefix(X, B), sufix(Y, B)), [_|_]).
```

Barem (se acordă maxim nota 3 pentru soluții care nu folosesc metapredicate):
1p – identificare X și Y a.î. A este o concatenare de X și Y
2p (1+1) – prefix(X, B), sufix(Y, B)
2p – corpul regulii este un singur metapredicat
1p – metapredicatul are template, goal și bag
1p – template și Bag corecte dpdv semantic
1p – goal corect dpdv semantic
1p – faptul că bag-ul conține cel puțin un element (pattern match sau verificare cu length)
1p – chiar dacă există mai multe descompuneri în X și Y, can_decompose se satisface o singură dată (un singur true)

10. Adăugăm următoarele definiții tipului Tournament de la exercițiul 6:

```
-- set = (game-uri câștigate, game-uri pierdute) - din perspectiva jucătorului curent
type Set = (Int, Int)
-- scor = listă de 2 sau 3 seturi (câștigă cine câștigă 2 seturi)
type Score = [Set]
-- performanța unui jucător la turneul cu numele name = listă de scoruri
-- (atâtea scoruri câte runde a jucat până a pierdut sau a câștigat turneul)
data TournamentRecord = TR Name [Score]
-- un jucător e descris prin numele său și o listă de performanțe la diverse turnee
data Player = P Name [TournamentRecord]
```

În rezolvarea fiecărei cerințe de mai jos este obligatoriu să folosiți **ori o funcțională, ori un list comprehension**:

a) Implementați funcția **playerWins :: Score -> Bool** care primește un scor și întoarce True dacă jucătorul (al cărui punctaj apare pe prima poziție în rezultatul fiecărui set) a câștigat acel meci, și False în caz contrar. **(10p)**
Ex: playerWins [(6,2), (6,7), (6,7)] => False

b) Implementați funcția **findTournamentByName :: Name -> [Tournament] -> Tournament**, care primește numele unui turneu și o listă de turnee și întoarce turneul cu numele respectiv (care se află, garantat, în listă). **(10p)**
Ex: findTournamentByName "Madrid" [T "Roma" [1,2,3,4,5], T "Madrid" [2,4,6,8,10]] => T "Madrid" [2,4,6,8,10])

c) Implementați funcția **playerPointsInTournament :: Player -> Tournament -> Points**, care primește un jucător și un turneu și calculează totalul punctelor acumulate de jucător în turneul respectiv. **(10p)**
Ex: pentru un jucător în al cărui TournamentRecord apare valoarea TR “Roma” [[(6,1), (6,1)], [(4,6), (7,6), (7,5)], [(3,6), (4,6)]] și turneul T "Roma" [1,2,3,4,5], funcția întoarce 1+2 = 3, pentru că jucătorul a câștigat primele 2 runde (acumulând punctele acordate pentru câștigarea lor), apoi a pierdut

Soluție:

```
playerWins :: Score -> Bool
playerWins score = length [ 1 | (player, opp) <- score, player > opp ] == 2

findTournamentByName :: Name -> [Tournament] -> Tournament
findTournamentByName name = head . filter (\(T n _) -> n == name)

playerPointsInTournament :: Player -> Tournament -> Points
playerPointsInTournament (P _ record) (T name points) = sum [ if won then p else 0 | (TR
n scores) <- record, n == name, (won, p) <- zip (map playerWins scores) points ]
```

Examen PP/2CB/14.06.23 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

Barem (la fiecare punct, se acordă maxim nota 2 pentru soluții care nu folosesc funcționale/list comprehensions):

a) 2p – accesarea valorilor din fiecare pereche de tip (game-uri castigate, game-uri pierdute)
   2p – condiția player > opp
   2p – numărarea seturilor câștigate
   2p – condiție de tip seturi câștigate == 2
   2p – asamblare corectă a tuturor componentelor

b) 3p – accesarea valorilor de tip nume turneu
   2p – condiția n == name
   2p – extragere turneu care respectă condiția
   3p – asamblare corectă a tuturor componentelor

c) 2p (1+1) – accesarea câmpurilor n și scores din record-ul unui player
   1p – condiția n == name
   2p – identificarea meciurilor pe care jucătorul le-a câștigat
   2p – corespondența între scorul unui meci și punctele acordate pentru victoria în acel meci
   1p – însumarea punctelor
   2p – asamblare corectă a tuturor componentelor