Examen PP/2CB/04.06.24 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. Identificați cel mai din stânga β-redex (precizând parametrul formal, corpul și parametrul efectiv) pentru λ-expresia:
   `(λx.λy.(y ((x x) y)) λx.λy.(y ((x x) y)))`

2. În Racket, se dau funcțiile f și g de mai jos:

```
(define (f L)
  (append (cdr L) (list (car L))))
(define (g L)
  (let helper ((L1 L) (L2 (f L)) (L3 (f (f L))))
    (cond ((null? L1) L1)
          ((even? (car L1)) (helper (cdr L1) (cdr L2) (cdr L3)))
          (else (cons (list (car L1) (car L2) (car L3))
                      (helper (cdr L1) (cdr L2) (cdr L3)))))))
```

   Ce rezultat întoarce apelul `(g '(1 2 3 4 5 6))`? **(2p)** Ce tip de recursivitate (stivă/coadă) generează acest apel? **(2p)** Rescrieți funcția g pentru a folosi celălalt tip de recursivitate. **(6p)**

3. În Racket, reimplementați funcția g de la exercițiul 2 folosind funcționale (fără a folosi deloc recursivitate explicită).

4. Se consideră implementat fluxul `odds` al numerelor naturale impare (1, 3, 5, 7 ...). În Racket, dați o implementare **implicită** pentru fluxul `odd-factorials` (1!, 3!, 5!, 7! ...), care să fie conformă cu schema de calcul de mai jos:

```
   1!  3!  5!  7!  9! ...
   3   5   7   9   11 ...
--------------------------- (o funcție de combinare pe care trebuie să o deduceți voi)
1!  3!  5!  7!  9!  11! ...
```

5. Sintetizați tipul Haskell pentru funcția `func` (nu sunt necesare explicații formale, dar sunt necesare explicații):

```
data BSTree a = Empty | Node a (BSTree a) (BSTree a) -- arbori binari de căutare
data List a = Nil | Cons a (List a)                  -- liste
func _ Empty = Empty
func f (Node [] left right) = Node Nil (func f left) (func f right)
func f (Node root left right) = Node (f root) (func f left) (func f right)
```

6. Fie în Haskell clasa BTree (definită mai jos) pentru arbori binari oarecare:

```
class BTree t where
  root :: t a -> Maybe a  -- rădăcina arborelui binar (folosim tipul Maybe
                          -- pentru că arborii vizi nu au rădăcină)
  nodes :: t a -> [a]     -- lista nodurilor din arbore, într-o ordine oarecare
```

Adăugați arborii binari de căutare de la exercițiul 5 la clasa BTree.

7. În logica cu predicate de ordinul întâi aduceți propoziția de mai jos la forma normal conjunctivă:
   `∀X.((câine(X) ∧ latră(X)) ⇒ ¬∃Y.mușcă(X,Y))`

8. Implementați, în Prolog, predicatul `take_drop(+N, +L, -Taken, -Dropped)`, care primește un număr N și o listă L și calculează lista primelor N elemente din L (în Taken) și lista L fără primele N elemente (în Dropped). **(5p)** Apoi implementați predicatul `group(+N, +L, -Grouped)` care primește un număr N și o listă L cu multiplu de N elemente și calculează o listă Grouped în care elementele listei L sunt grupate câte N (în aceeași ordine). (ex: pentru lista [1,2,3,4,5,6] și N=2, se obține [[1,2], [3,4], [5,6]]). **(5p)**

9. În Prolog, considerând implementate predicatele unare cond1 și cond2 care reușesc dacă argumentul lor satisface condițiile cond1, respectiv cond2, implementați – folosind **metapredicate** și nefolosind recursivitate explicită – predicatul check(+L), care primește o listă și reușește dacă se respectă simultan următoarele:
   (1) toate elementele din L satisfac cel puțin o condiție dintre cond1 și cond2
   (2) peste jumătate din elementele din L satisfac atât cond1 cât și cond2.

Examen PP/2CB/04.06.24 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

10. Pentru tipul BSTree a și clasa BTree de la exercițiile 5 și 6:

(a) Adăugați funcția arcs atât la definiția clasei BTree **(3p)** cât și la instanțierea efectuată la exercițiul 6 (puteți considera exercițiul 6 rezolvat și să adăugați doar această funcție). **(7p)** Funcția arcs va primi un arbore binar și va întoarce toate arcele arborelui – o listă completă de perechi (nod părinte, nod copil).

(b) **(5p)** Implementați funcția `myIterate :: (a -> Maybe a) -> a -> [a]` care primește o funcție `f :: a -> Maybe a` și un element de start `x :: a`, și creează o listă care:
- pe prima poziție îl are pe x
- pe a doua poziție are rezultatul de tip a produs de f x, dacă acesta există
- pe a treia poziție are rezultatul de tip a produs de f asupra elementului anterior, dacă acest rezultat există
- ... etc.

În momentul în care f pe argumentul primit produce un Nothing, se încheie și generarea listei.

(c) **(5p)** Folosind funcția `myIterate`, implementați funcția `path :: (Eq a, BTree t) => t a -> a -> [a]`, care primește un arbore binar `tree` și un nod `x` și întoarce lista nodurilor de pe calea de la rădăcina lui `tree` la nodul `x` (primul nod din cale va fi rădăcina lui `tree`, ultimul va fi chiar `x`).

(d) **(10p)** Implementați funcția `lowestCommonAncestor :: (Eq a, BTree t) => t a -> a -> a -> a`, care primește un arbore binar `tree` și două noduri `x` și `y` și întoarce cel mai adânc strămoș comun din `tree` al celor 2 noduri (se consideră că orice nod este și propriul său strămoș).