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)))`

Soluție: Întreaga expresie este un β-redex. Am colorat corespunzător parametrul formal, corpul și parametrul efectiv.
Barem: 2p parametru formal, 4p corp, 4p parametru efectiv

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)**

Soluție:

```
(g '(1 2 3 4 5 6)) => '((1 2 3) (3 4 5) (5 6 1))
```

Recursivitate pe stivă.

```
(define (g L)
  (let helper ((L1 L) (L2 (f L)) (L3 (f (f L))) (acc '())) ;; cu rec pe coadă
    (cond ((null? L1) (reverse acc))
          ((even? (car L1)) (helper (cdr L1) (cdr L2) (cdr L3) acc))
          (else (helper (cdr L1) (cdr L2) (cdr L3)
                        (cons (list (car L1) (car L2) (car L3)) acc))))))
```

Barem:
a) 2p
b) 2p
c) 2p - funcție ajutătoare sau named let cu parametri corespunzători
  0.5p - condiții corecte (null?, even?)
  1p - întoarce acc pe cazul de bază
  0.5p - rezultat în ordine corectă (reverse acc sau append la sfârșit)
  2p - apeluri recursive cu parametri actualizați corect

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

Soluție:

```
(define (g-func L)
  (filter (compose odd? car)
          (map list L (f L) (f (f L)))))
```

Barem:
4p - map sau fold aplicat corespunzător pe o funcție și 3 liste (și acc pt fold)
1p - funcție de combinare corectă (list sau echivalent)
1p - liste corecte ca argumente pentru map sau fold
2p - filter aplicat corespunzător pe un predicat și o listă
1p - predicat corect
1p - filter aplicat pe rezultatul map (sau fold), nu invers

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

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! ...
```

Soluție (se consideră definit stream-zip-with):

```
(define odd-factorials
  (stream-cons 1
               (stream-zip-with (λ (f n) (* f n (sub1 n)))
                                odd-factorials
                                (stream-rest odds))))
```

Barem:

4p - stream-zip-with aplicat corespunzător pe o funcție și două fluxuri
2p - funcție de combinare corectă
2p - fluxuri corecte (1p odd-factorials, 0.5p odds, 0.5p stream-rest)
2p - primul element dat "manual"

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)
```

Soluție (+Barem):

func este o funcție de 2 argumente => func :: a -> b -> c (1p)
Al doilea argument este obținut prin aplicarea constructorilor Empty sau Node, deci b = BSTree d. (1p)
În al doilea argument al lui func, Node este aplicat pe [] ca prim argument, deci d = [e]. (1.5p)
În al doilea argument al lui func, Node este aplicat pe root ca prim argument, deci root :: [e] (1p)
func este aplicat pe f ca prim argument, deci f :: a
f este aplicat pe root, deci f este o funcție al cărei argument este de tipul [e], adică a = [e] -> g (1p)
Rezultatul este obținut prin aplicarea constructorilor Empty sau Node, deci c = BSTree h (1p)
În acest rezultat, Node este aplicat pe Nil ca prim argument, deci h = List i (1.5p)
În acest rezultat, Node este aplicat pe (f root) ca prim argument, deci (f root) :: List i, adică a = [e] -> List i (1p)
Restul aplicărilor este consistent cu constrângerile de până acum și nu produce noi constrângeri.

=> func :: ([e] -> List i) -> BSTree [e] -> BSTree (List i) (1p)

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.

Soluție:

```
instance BTree BSTree where
  root (Node r _ _) = Just r
  root _ = Nothing

  nodes (Node r left right) = r : nodes left ++ nodes right
  nodes _ = []
```

Barem:

2p - instanțiere cu BSTree, nu cu (BSTree a) sau altceva

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

4p – implementare root (1p pattern matching corect, 1p folosire tipul Maybe, 2p extragere corectă root)

4p – implementare nodes (1p pattern matching corect, 1p definiția pentru Empty, 2p definiția pentru Node)

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))`

Soluție (+Barem):

Elimină implicațiile: `∀X.(¬(câine(X) ∧ latră(X)) ∨ ¬∃Y.mușcă(X,Y))` (2p)

Mută negațiile spre interior: `∀X.(¬câine(X) ∨ ¬latră(X) ∨ ∀Y.¬mușcă(X,Y))` (3p)

Mută cuantificatorii la început: `∀X.∀Y.(¬câine(X) ∨ ¬latră(X) ∨ ¬mușcă(X,Y))` (2p)

Elimină cuantificatorii: `¬câine(X) ∨ ¬latră(X) ∨ ¬mușcă(X,Y)` (2p)

Transformă în clauze: `{¬câine(X), ¬latră(X), ¬mușcă(X,Y)}` (1p)

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)**

Soluție:

```
take_drop(0, L, [], L).
take_drop(N, [X|Xs], [X|T], D) :- N > 0, N1 is N-1, take_drop(N1, Xs, T, D).

group(_, [], []).
group(N, L, [Group|Gs]) :- take_drop(N, L, Group, Dropped), group(N, Dropped, Gs).
```

Barem:

a) 2p - cazul de bază

3p - cazul general (0.5p reguli mutual exclusive – în clar condiția N>0, 0.5p scădere cu is, 1p apel recursiv, 1p adăugarea corectă a lui X în lista Taken rezultat)

b) 2p - cazul de bază

3p - cazul general (1p calcul grup, 1p apel recursiv, 1p adăugarea lui Group în lista rezultat)

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.

Soluție:

```
check(L) :-
  forall(member(X, L), (cond1(X) ; cond2(X))),
  findall(_, (member(X, L), cond1(X), cond2(X)), R),
  length(R, LenR), length(L, LenL), 2 * LenR > LenL.
```

Barem:

4p - metapredicat pentru "cond1 sau cond2" (1p număr și tip de argumente corecte, 1p member, 2p verificare corectă)

4p - metapredicat pentru "cond1 și cond2" (1p număr și tip de argumente corecte, 1p member, 2p verificare corectă)

2p - verificare cardinalitate (1p "toate", 1p "peste jumătate")

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).

Soluție:

```
class BTree t where
  …
  arcs :: t a -> [(a,a)]

instance BTree BSTree where
  …
  arcs (Node r left right) = [ (r,x) | Just x <- [root left, root right] ] ++ arcs left
++ arcs right
  arcs _ = []

myIterate f x = case f x of
  Nothing -> [x]
  Just fx -> x : myIterate f fx

getParent lst node = if null p then Nothing else Just (head p)
  where p = [ x | (x, y) <- lst, y == node ]
path tree node = reverse $ myIterate (getParent (arcs tree)) node

lowestCommonAncestor tree node1 node2 = fst $ last $ takeWhile (\(x,y) -> x == y) $ zip
(path tree node1) (path tree node2)
```

Barem:

a) 3p - signatura arcs (1p t a, 1p ->, 1p [(a,a)])
  1p - cazul de bază (pt Empty)
  1p - pattern matching corect
  1p - apelurile recursive pe left și right
  1p - extragere root-uri left și right
  1p - creare listă de perechi
  1p - concatenarea tuturor termenilor
  1p - acoperirea tuturor situațiilor (subarbori vizi și nevizi)
b) 2p - cazul de bază (1p testul f x == Nothing, 1p rezultat)
  3p - cazul general (1p adăugare x, 1p apel recursiv corect, 1p extragere element din Just)
c) 3p - funcție de extragere părinte (1p retur corect de tip Maybe, 2p extragere corectă părinte)
  2p - extragere cale (0.5p ordine corectă, 0.5p apel myIterate, 1p corectitudine argumente myIterate)
d) 3p - extragere path-uri pentru node1 și node2
  5p - parcurgere path-uri cât timp sunt identice
  2p - extragere nod (lowestCommonAncestor)