Varianta C – Examen PP (2020)

1. `(λx y z.corp a b c)` este forma prescurtată de a scrie `(((λx.λy.λz.corp a) b) c)`.

   Fie următoarele definiții în Calcul Lambda:

   `null = λx.true`

   `cons = λx y z.(z x y)`

   și fie operatorul pe liste:

   `op = λL.(L λx y.false)`

   Descrieți pas cu pas (nu efectuați mai multe β-reduceri deodată) comportamentul lui `op` pe valori de tip listă, astfel încât să puteți conchide la final că `op` se comportă ca unul dintre operatorii pe liste cunoscuți (și precizați care este operatorul a cărui identitate a fost ascunsă prin folosirea numelui "`op`").

   Soluție:

   `(op null) = (λL.(L λx y.false) λx.true)` $\rightarrow$ `(λx.true λx y.false)` $\rightarrow$ `true`

   `(op (cons a b)) = (op (λx y z.(z x y) a b))` $\rightarrow$ `(op λz.(z a b))`

   `(op λz.(z a b)) = (λL.(L λx y.false) λz.(z a b))` $\rightarrow$ `(λz.(z a b) λx y.false)` $\rightarrow$ `(λx y.false a b)` $\rightarrow$ `false`

   Concluzie: `op` este `null?`

2. Ce tip de recursivitate are implementarea de mai jos?

```
(define (t x)
    (cond ((or (< x 0) (even? x)) 0)
        ((< x 5) (quotient x 2))
        (else (+ (t (- x 2)) (t (- x 4))))))
```

   Definiți (tot în Racket) cât mai eficient funcția `tt` care are același efect cu `t`, dar folosește un alt tip de recursivitate.

   Care este noul tip de recursivitate?

   Soluție:

   Recursivitate arborescentă – o vom transforma în recursivitate pe coadă.

```
(define (tt x) ;; e fix fibonacci, doar că din 2 în 2
   (let t ((x x) (a 0) (b 1))
      (cond ((or (< x 0) (even? x)) 0)
         ((= x 1) a)
         ((= x 3) b)
         (else (t (- x 2) b (+ a b))))))
```

3. **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.

   Se dă o listă `L` care conține numere și/sau liste de numere (maxim un nivel de imbricare). Să se definească, în Racket, funcția `longest-list` care întoarce lista de lungime maximă din input - fie întreg `L`, fie una din listele interioare lui `L`.

   ex: `(longest-list '(0 1 2 (3 4 3 4 3 4 3 4 3 4) 5 (6) 7))` va întoarce `'(3 4 3 4 3 4 3 4 3 4)` - pt ca are lungimea 10

   `(longest-list '(0 1 2 (3 4 3 4 3 4) 5 (6) 7))` va întoarce `'(0 1 2 (3 4 3 4 3 4) 5 (6) 7))` - pt ca are lungimea 7

   Soluție:

```
(define (longest-list L)
   (let ((inner-lists (filter list? L)))
      (foldr (lambda (L acc) (if (> (length L) (length acc)) L acc))
         L
         inner-lists)))
```

4. Definiți în Racket fluxul coeficienților binomiali:

   `'((1) (1 1) (1 2 1) (1 3 3 1) (1 4 6 4 1) (1 5 10 10 5 1) ...)`

   Soluție:

```
(define binomial-stream
   (stream-cons '(1)
      (stream-map (lambda (L) (let ([M (cons 0 L)]) (map + M (reverse M))))
         binomial-stream)))
```

5. Sintetizați în Haskell, pas cu pas, tipul funcției `f`:

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

   Soluție:

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

6. Scrieți o instanță posibilă a clasei de mai jos, conținând o implementare **neconstantă** a funcției `f`.

```
class MyClass c where
    f :: Num a => c a -> c a -> c a
```

   Soluție:

```
instance MyClass Maybe where
    f (Just x) (Just y) = Just $ x + y
    f _ _ = Nothing
```

7. Folosiți pentru scriere `NOT()`, `AND` și `OR` ca operatori logici și `ORICARE` și `EXISTĂ` drept cuantificatori. Scrieți cu caps ca să nu existe confuzii cu limbajul obișnuit. Puteți prescurta numele predicatelor.

   Știind că "Unde intră soarele pe fereastră, nu intră doctorul pe ușă.", că `doctor(hipocrate)`, că `locuință(apartamentul_2)`, și că `intră_pe_ușă(hipocrate, apartamentul_2)`, dorim să demonstrăm că `NOT(intră_soarele_pe_vreo_fereastră_din(apartamentul_2))`.

   a) Traduceți proverbul în LPOI (FOL) utilizând numai predicatele menționate în problemă.

   b) Enumerați toate clauzele necesare demonstrației (inclusiv negarea concluziei), ilustrând pe scurt cum ați ajuns la ele.

   Obs: nu se cere demonstrația propriu-zisă.

   Soluție:

   a) $\forall$`X`.$\forall$`Y.locuință(X)` $\wedge$ `intră_soarele_pe_vreo_fereastră_din(X)` $\wedge$ `doctor(Y)` $\Rightarrow$ $\neg$`intră_pe_ușă(Y, X)`

   b) Clauze:

      (1) `doctor(hipocrate)`

      (2) `locuință(apartamentul_2)`

      (3) `intră_pe_ușă(hipocrate, apartamentul_2)`

      (4) `intră_soarele_pe_vreo_fereastră_din(apartamentul_2)`

      (5) $\neg$`locuință(X)` $\vee$ $\neg$`intră_soarele_pe_vreo_fereastră_din(X)` $\vee$ $\neg$`doctor(Y)` $\vee$ $\neg$`intră_pe_ușă(Y, X)`

8. Implementați predicatul `a(+L1, +L2, +L3, -L)` care primește 3 liste și produce în `L` o listă de tripluri cu elementele ce corespund din cele 3 liste. Lista `L` are lungimea celei mai scurte liste dintre `L1`, `L2` și `L3`. Exemplu: `a([1, 2], [a, b, c], [x, y, z, t], X)` leagă `X` la `[(1, a, x), (2, b, y)]`.

   Explicați cum funcționează implementarea.

   Soluție:

```
a([], _, _, []).
a(_, [], _, []).
a(_, _, [], []).
a([H1|L1], [H2|L2], [H3|L3], [(H1, H2, H3) | L123]) :- a(L1, L2, L3, L123).
```

9. Folosiți unul sau mai multe dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `secondMin(+L, -X)` care găsește al doilea cel mai mic element din lista `L` (care conține cel puțin 2 elemente și nu conține duplicate). Nu utilizați recursivitate explicită. Explicați cum funcționează soluția.

   Soluție:

```
secondMin(L, X) :- member(X, L), findall(Y, (member(Y, L), Y < X), [_]).
```

10. Problema urmărește reprezentarea unor propoziții din logica propozițională și implementarea unei transformări asupra acestora.

    Utilizați următorul schelet:

    - (8p) Elaborați reprezentarea propozițiilor logice în forma unui tip de date Haskell. Propozițiile vor putea consta în:
      - propoziții simple (exemplu: p, q, r etc.)
      - negații (exemplu: $\neg p$)
      - conjuncții (exemplu: $p \wedge q$)
      - disjuncții (exemplu: $p \vee q$).

      ```
      data Prop
          = Undefined
      ```

    - (2p) Pornind de la reprezentarea de mai sus, traduceți propoziția $\neg(p \wedge (\neg q \vee r))$.

      ```
      prop :: Prop
      prop = undefined
      ```

    - (10p) Instanțiați clasa `Show` cu tipul `Prop`. Reprezentarea sub formă de șir a propoziției prop ar trebui să fie cea din cerința de mai sus.

      ```
      instance Show Prop where
          show = undefined
      ```

    - (10p) Implementați funcția `processNeg`, care urmărește introducerea negațiilor în paranteze. De exemplu:

      `show (processNeg prop)` va furniza $(\neg p \vee (q \wedge \neg r))$

      ```
      processNeg :: Prop -> Prop
      processNeg = undefined
      ```

    Soluție:

```
-- 1.1
data Prop
    = Simple Char
    | Neg Prop
    | And Prop Prop
    | Or Prop Prop
-- 1.2
prop :: Prop
prop = Neg $ Simple 'p' `And` (Neg (Simple 'q') `Or` Simple 'r')
```

```haskell
-- 2
instance Show Prop where
    show (Simple c) = [c]
    show (Neg p) = "~" ++ show p
    show (And p1 p2) = "(" ++ show p1 ++ " ^ " ++ show p2 ++ ")"
    show (Or p1 p2) = "(" ++ show p1 ++ " v " ++ show p2 ++ ")"
-- 3
processNeg :: Prop -> Prop
processNeg p@(Simple _) = p
processNeg (Neg p) = case p of
    Simple _  -> Neg p
    Neg p'    -> processNeg p'
    And p1 p2 -> processNeg $ Or (Neg p1) (Neg p2)
    Or p1 p2  -> processNeg $ And (Neg p1) (Neg p2)
processNeg (And p1 p2) = And (processNeg p1) (processNeg p2)
processNeg (Or p1 p2) = Or (processNeg p1) (processNeg p2)
```