Varianta D – 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 următoarea definiție în Calcul Lambda:

   `cons = λx y z.(z x y)`

   și fie operatorul pe liste nevide:

   `op = λL.(L λx y.y)`

   Descrieți pas cu pas (nu efectuați mai multe β-reduceri deodată) comportamentul lui `op` pe valori de tip listă nevidă, 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 (cons a b)) = (op (λx y z.(z x y) a b)) → (op λz.(z a b))`

   `(op λz.(z a b)) = (λL.(L λx y.y) λz.(z a b)) → (λz.(z a b) λx y.y) → (λx y.y a b) → b`

   Concluzie: `op` este `cdr`

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

```
(define (w x)
    (let h ((x x) (r 0))
        (if (null? x)
            (* 2 r)
            (add1 (h (cdr x) (if (even? (car x)) r (add1 r)))))))
```

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

   Precizați pe scurt care modificare din cod a dus la modificarea tipului de recursivitate.

   Soluție:

   Recursivitate pe stivă – o vom transforma în recursivitate pe coadă.

```
(define (ww x)
   (let h ((x x) (r 0))
      (if (null? x)
         r
         (h (cdr x) (if (even? (car x)) (add1 r) (+ r 3)))))) ;; înainte
         ↪  aveam add1 pe apelul recursiv (stivă), acum apelul recursiv
         ↪  este la coadă, toate adunările se fac asupra acumulatorului r
```

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` de numere. Să se definească, în Racket, funcția `count-smaller` care întoarce o listă de perechi de forma `(număr_n_din_L . câte_valori_din_L_sunt_mai_mici_ca_n)`. Atenție, fiecare număr din `L` trebuie să apară o singură dată în partea din stânga a perechilor!

   ex: `(count-smaller '(1 2 3 2 4 3 1 2))` va întoarce `'((4 . 7) (3 . 5) (2 . 2) (1 . 0))` - sau aceleași 4 perechi în orice altă ordine (ordinea **NU** contează), pentru că în `L` sunt 7 numere mai mici ca 4, 5 mai mici ca 3, 2 mai mici ca 2, și niciunul mai mic ca 1.

   Soluție:

```
(define (count-smaller L)
   (let ((no-dups (foldl (lambda (n acc) (if (member n acc) acc (cons n
   ↪  acc))) '() L)))
      (map (lambda(n) (cons n (length (filter (lambda (x) (< x n)) L))))
      ↪  no-dups)))
```

4. Se dau în Racket două fluxuri de liste, `s1` și `s2`. Definiți fluxul ale căror elemente se obțin prin intersectarea listelor de pe aceeași poziție din `s1` și `s2`.

   Exemplu: `s1 = '((1 2) (3 4 5) ...), s2 = '((2 3) (1 4 5) ...), rezultat = '((2) (4 5) ...)`

   Soluție:

```
(define result
  (stream-zip-with (lambda (L1 L2) (filter (lambda (e) (member e L1)) L2))
  ↪  s1 s2))
```

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

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

   Soluție:

```
f :: a -> b -> c
x :: a
y :: b
a = (b, b)
b = [a]
a = ([a], [a])
eroare
```

6. Supraîncărcați în Haskell operatorul de comparație pe liste, astfel încât o listă să fie mai mică sau egală cu alta, dacă toate elementele din prima listă sunt mai mici sau egale cu toate elementele din a doua. Spre exemplu, `[2, 3, 1] <= [5, 4, 3]`, dar nu avem că `[2, 4] <= [3, 5]`.

   Soluție:

```
instance Ord a => Ord [a] where
    xs <= ys = maximum xs <= minimum ys
```

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.

   Considerăm proverbul "Cine fură azi un ou, mâine va fura un bou.", și predicatele:

```
fură_azi(Cine, Ce).
fură_mâine(Cine, Ce).
bou(Ce).
ou(Ce).
```

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

   b) Traduceți propoziția în LPOI în formă clauzală (FNC).

   Soluție:

   a) $\forall$`x.`$\forall$`y.ou(x)` $\wedge$ `fură_azi(y, x)` $\Rightarrow$ $\exists$`z.bou(z)` $\wedge$ `fură_mâine(y, z)`

   b) Clauze:

   (1) $\neg$`ou(x)` $\vee$ $\neg$`fură_azi(y, x)` $\vee$ `bou(bou_furat(y))`

   (2) `ou(x)` $\vee$ $\neg$`fură_azi(y, x)` $\vee$ `fură_mâine(y, bou_furat(y))`

8. Implementați predicatul `mul(+Lin, -Lout)` care primește o listă de numere ca prim argument și leagă al doilea argument la lista acelor numere din prima listă care sunt divizibile cu toate celelalte numere care le urmează.

   Exemplu: `mul([24, 4, 12, 6, 3, 2], X)` leagă `X` la `[24, 12, 6, 2]`.

   Explicați cum funcționează implementarea.

Soluție:

```
check(_, []).
check(X, [H|T]) :- mod(X, H) =:= 0, check(X, T).
mul([], []).
mul([H|LIn], [H|LOut]) :- check(H, LIn), !, mul(LIn, LOut).
mul([_|LIn], LOut) :- mul(LIn, LOut).
```

9. Folosiți unul sau mai multe dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `secondMax(+L, -X)` care găsește al doilea cel mai mare 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 arborilor **ȘI-SAU** și operații de bază asupra acestora.

    Un arbore **ȘI-SAU** este un arbore următoarele caracteristici:

    - nodurile care nu sunt frunze sunt de tipul **ȘI** (**AND**) sau de tipul **SAU** (**OR**) și pot avea oricâți copii.
    - orice nod este o etichetă care poate fi **SUCCESS**, **FAILURE**, sau **UNKNOWN**.
    - (10p) Elaborați reprezentarea arborilor ȘI-SAU. La nevoie, puteți utiliza și alte tipuri auxiliare.

      ```
      data AndOrTree
          = Undefined
      ```

    - (10p) Instanțiați clasa `Show` cu tipul `AndOrTree`. Reprezentarea sub formă de șir a unui arbore va fi de forma următoare, în care frunzele sunt reprezentate prin eticheta lor, iar nodurile **ȘI** și **SAU** sunt reprezentate prin tipul și eticheta lor:

      ```
      (OR|UNKNOWN (AND|UNKNOWN FAILURE (OR|UNKNOWN UNKNOWN SUCCESS) SUCCESS)
            (AND|UNKNOWN SUCCESS (OR|UNKNOWN FAILURE SUCCESS)))
      ```

      ```
      instance Show AndOrTree where
          show = undefined
      ```

    - (10p) Implementați funcția `compute` care primește un arbore **ȘI-SAU** și produce un arbore **ȘI-SAU** în care au fost calculate etichetele pentru noduri, după regulile:
      - un nod **ȘI** are eticheta **SUCCESS** dacă toți copiii au eticheta **SUCCESS**, altfel **FAILURE**.
      - un nod **SAU** are eticheta **SUCCESS** dacă cel puțin un copil are eticheta **SUCCESS**, altfel **FAILURE**.
      - toate frunzele rămân nemodificate.

      De exemplu, pentru arborele reprezentat ca în exemplul de la punctul anterior, rezultatul funcției compute este afișat ca:

      ```
      (OR|SUCCESS (AND|FAILURE FAILURE (OR|SUCCESS UNKNOWN SUCCESS) SUCCESS)
           (AND|SUCCESS SUCCESS (OR|SUCCESS FAILURE SUCCESS)))
      ```

      ```
      compute :: AndOrTree -> AndOrTree
      compute = undefined
      ```

    Soluție:

    ```
    -- 1
    data AndOrTree
       = AND Tag [AndOrTree]
       | OR Tag [AndOrTree]
       | Leaf Tag
    data Tag = SUCCESS | FAILURE | UNKNOWN deriving (Show, Eq)
    ```

```haskell
-- 2
instance Show AndOrTree where
   show (AND tag children) = intern "AND" tag children
   show (OR tag children) = intern "OR" tag children
   show (Leaf tag) = show tag

intern t tag chd = "(" ++ t ++ "|" ++ show tag ++ concatMap ((" " ++) .
↪  show) chd ++ ")"

-- 3
getTag (AND t _) = t
getTag (OR t _) = t
getTag (Leaf t) = t

compute :: AndOrTree -> AndOrTree
compute l@(Leaf tag) = l
compute (AND tag children) = AND (if all ((==SUCCESS) . getTag) chd then
↪  SUCCESS else FAILURE) chd
   where chd = map compute children
compute (OR tag children) = OR (if any ((==SUCCESS) . getTag) chd then
↪  SUCCESS else FAILURE) chd
   where chd = map compute children
```