

Varianta C – Examen PP (2020)

1. $(\lambda x y z. \text{corp } a \text{ } b \text{ } c)$ este forma prescurtată de a scrie $(((\lambda x. \lambda y. \lambda z. \text{corp } a) \text{ } b) \text{ } c)$.

Fie următoarele definiții în Calcul Lambda:

```
null =  $\lambda x. \text{true}$ 
```

```
cons =  $\lambda x y z. (z \text{ } x \text{ } y)$ 
```

și fie operatorul pe liste:

```
op =  $\lambda L. (L \text{ } \lambda x y. \text{false})$ 
```

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").

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?

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

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) ...)
```

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

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

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
```

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))`.

- Traduceți proverbul în LPOI (FOL) utilizând numai predicatele menționate în problemă.
- Enumerați toate clauzele necesare demonstrației (inclusiv negarea concluziei), ilustrând pe scurt cum ați ajuns la ele.

Obs: nu se cere demonstrația propriu-zisă.

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.
9. Folosiți unul sau mai multe dintre predicatul `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.
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: `¬p`)
 - conjuncții (exemplu: `p ∧ q`)
 - disjuncții (exemplu: `p ∨ 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 (¬p ∨ (q ∧ ¬r))

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