

Precizări:

- Fiecare subiect are câte 10p.
- Este suficientă rezolvarea a 10 subiecte din cele 12 pentru nota maximă.

1. Ce proprietăți trebuie să îndeplinească expresia  $F$ , astfel încât expresia  $E$  să fie reductibilă? Justificați!

$$E \equiv (((\lambda x.x \ \lambda x.x) \ \Omega) \ (F \ \Omega))$$
$$\Omega \equiv (\lambda x.(x \ x) \ \lambda x.(x \ x))$$

*Soluție.* .

Secvența de reducere stânga-dreapta, a lui  $E$ , nu se termină. Conform teoremei normalizării, nicio altă secvență nu se termină.  $\square$

2. Fie o listă Scheme, precum `'(1 (2 3) () 4)`, ce poate conține alte liste drept elemente. Calculați numărul acestor liste-element, întrebuițând **funcționale** și **fără** a utiliza recursivitate explicită. Pentru exemplul de mai sus, rezultatul este 2. Solutiile care nu respectă cele două constrângeri **nu** vor fi punctate!

*Soluție.* .

```
(length (filter list? '(1 (2 3) () 4)))
```

$\square$

3. La ce se evaluează ultima expresie, în Scheme? Cum s-ar modifica rezultatul dacă `f` ar fi redefinită și aplicată în formă *curried*? Justificați!

```
1 (define f
2   (lambda (x y)
3     (+ x y)))
4
5 (f 1 (let ([x 2]) 3))
```

*Soluție.* .

Rezultatul este 4 în ambele situații, deoarece domeniile de vizibilitate ale celor două variabile `x` sunt izolate.  $\square$

4. Definiți obiectul `x` în Scheme, astfel încât expresia `(eq? (car (x)) (cdr (x)))` să se evalueze la #t.

*Soluție.* .

```
1 (define x (lambda () (cons x x)))
```

$\square$

5. Căți parametri se vor calcula în procesul de evaluare a expresiei Haskell de mai jos? Justificați!

```
let f x y z = x + y - z in f (1 + 2) (2 + 3)
```

*Soluție.* .

Niciunul, deoarece evaluarea este lenesă, iar valoarea este o închidere funcțională.  $\square$

6. Sintetizați tipul funcției Haskell de mai jos. Justificați!

$$f \ x \ y = x \ (f \ y \ x)$$

*Soluție.* .

```
1  f :: a -> b -> c
2  x :: d -> e
3  a = d -> e
4  d = c
5  e = c
6  b = a
7  f :: (c -> c) -> (c -> c) -> c
```

$\square$

7. Supraîncărcați în Haskell operatorul de verificare a egalității funcțiilor care acceptă numere drept prim parametru. Acestea vor fi considerate egale dacă valorile lor coincid în punctul 5.

*Soluție.* .

```
1  instance (Num a, Eq b) => Eq (a -> b) where
2      f == g = (f 5) == (g 5)
```

Barem:

- 5p: instanțiere.
- Câte 2,5p: fiecare din cele două componente ale contextului.

$\square$

8. Definiți în Haskell fluxul sirurilor "", "a", "aa", "aaa", ....

*Soluție.* .

```
1  strings = "" : (map ('a' :) strings)
```

$\square$

9. Transformați în formă normală conjunctivă (FNC) următoarea propoziție:

$$\forall x. (om(x) \Rightarrow \exists y. (inima(y) \wedge are(x, y))).$$

*Soluție.* .

$$\begin{aligned} & \forall x. (om(x) \Rightarrow \exists y. (inima(y) \wedge are(x, y))) \\ & \forall x. (\neg om(x) \vee \exists y. (inima(y) \wedge are(x, y))) \\ & \forall x. \exists y. (\neg om(x) \vee (inima(y) \wedge are(x, y))) \\ & \forall x. (\neg om(x) \vee (inima(f_y(x)) \wedge are(x, f_y(x)))) \\ & \neg om(x) \vee (inima(f_y(x)) \wedge are(x, f_y(x))) \\ & (\neg om(x) \vee inima(f_y(x))) \wedge (\neg om(x) \vee are(x, f_y(x))) \\ & \{\neg om(x), inima(f_y(x))\}, \{\neg om(x), are(x, f_y(x))\} \end{aligned}$$

□

10. Scrieți un program Prolog, care realizează *insertion sort*.

*Soluție.* .

```
1 insert([ ], [ ]).
2 insert([ X | T ], [ X , H | T ]) :- X <= H, ! .
3 insert([ X | T ], [ H | L ]) :- insert(X, T, L) .
4
5 isort([ ], [ ]).
6 isort([ H | T ], S) :- isort(T, L), insert(H, L, S) .
```

□

11. Scrieți un algoritm Markov, care dublează toate simbolurile de pe bandă. Alfabetul de bază este  $A_b = \{0, 1\}$ . De exemplu, pentru banda inițială 101100, se obține 110011110000.

*Soluție.* .

În programul de mai jos, g este o variabilă generică, iar a, una de lucru.

```
1 Double() ; Ab g
2      ag -> gga
3      a -> .
4      -> a
5 end
```

□

12. Scrieți un program CLIPS, care, la sfârșitul execuției, lasă faptul (`(included)`) în baza de cunoștințe, dacă mulțimea A este inclusă în mulțimea B. De exemplu, pentru faptele `(A 1 3 5)` și `(B 1 2 3 4 5)`, răspunsul va fi prezent, în timp ce, pentru faptele `(A 1 3 5 6)` și `(B 1 2 3 4 5)`, nu va fi.

*Soluție.* .

```
1 (deffacts sets
2   (A 1 3 5 6)
3   (B 1 2 3 4 5))
4
5 (defrule init
6   =>
7   (assert (included)))
8
9 (defrule check
10   ?i <- (included)
11   (A $? ?x $?)
12   (not (B $? ?x $?)))
13   =>
14   (retract ?i))
```

□

Precizări:

- Fiecare subiect are câte 10p. **Punctajul NU se acordă în absență justificării răspunsului!**
- Este suficientă rezolvarea a **10** subiecte din cele 12 pentru nota maximă.
- Punctajul suplimentar reprezintă **bonus**, ce poate compensa punctajul de pe parcurs.

1. Fie următoarele definiții în calculul lambda:

$$\begin{aligned} \text{one} &\equiv \lambda f. f \\ \text{increment} &\equiv \lambda n. \lambda f. (f \ (n \ f)). \end{aligned}$$

Descrieți evaluarea pas cu pas a expresiilor (5p + 5p):

$$\begin{aligned} (\text{increment } \text{one}) \\ (\text{increment } (\text{increment } \text{one})). \end{aligned}$$

*Soluție.* .

$$\begin{aligned} (\text{increment } \text{one}) &\rightarrow \lambda f. (f \ (\lambda f. f \ f)) \rightarrow \lambda f. (f \ f) \\ (\text{increment } (\text{increment } \text{one})) &\rightarrow \lambda f. (f \ (\lambda f. (f \ f) \ f)) \rightarrow \lambda f. (f \ (f \ f)). \end{aligned}$$

□

2. Fie funcția Racket de mai jos:

```
1 (define (f L x)
2   (if (null? L)
3     x
4     (+ (car L)
5       (f (cdr L) (+ x (car L))))))
```

- (5p) Ce calculează  $(f \ L \ 0)$ ?
- (5p) Ce tip de recursivitate utilizează  $f$ ?

*Soluție.* .

- Dublul sumei elementelor listei  $L$ , deoarece fiecare element se adună atât pe avans, cât și pe revenire.
- Pe stivă, deoarece una din însumările cu  $(\text{car } L)$  se face pe revenire. □

3. Fie următoarea reprezentare în Haskell a unui arbore binar:

```
1 data Tree a
2   = Empty
3   | Node a (Tree a) (Tree a)
```

Pornind de la definiția funcționalei  $\text{foldr}$ , rezolvați subpunctele de mai jos:

```
1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr f acc [] = acc
3 foldr f acc (h : t) = f h $ foldr f acc t
```

- (a) (4p) Care este tipul funcționalei `foldT`, care extinde `foldr` pentru „împăturirea” unui arbore binar?
- (b) (4p) Care este implementarea acesteia?
- (c) (2p) Pe baza lui `foldT`, definiți o funcție pentru însumarea tuturor cheilor dintr-un arbore numeric.

*Soluție.* .

```

1 foldT :: (a -> b -> b -> b) -> b -> Tree a -> b
2 foldT _ acc Empty = acc
3 foldT f acc (Node key left right) =
4     f key (foldT f acc left) (foldT f acc right)
5
6 sumT :: Num a => Tree a -> a
7 sumT = foldT (\x y z -> x + y + z) 0

```

□

4. Fie următoarea secvență în Racket:

```

1 (define f
2   (lambda (x)
3     (lambda (y)
4       (+ x y))))
5
6 (define x 0)
7
8 (define g (f x))
9 (g 10)
10
11 (define x 1)
12 (g 10)

```

Care sunt valorile celor două aplicații  $(g\ 10)$ ? Justificați utilizând modelul contextual de evaluare (închideri și contexte).

*Soluție.* .

Ambele sunt 10, deoarece, la evaluarea aplicației  $(f\ x)$ , valoarea lui  $x$  din acel moment (0) este salvată în închiderea construită:  $g \leftarrow \langle \lambda y. (+\ x\ y); x \leftarrow 0 \rangle$ . □

5. Construiți în Haskell fluxul compunerilor unei funcții cu ea însăși, de ori câte ori. De exemplu, `compositions (+ 1) = [ (+ 1), (+ 1) . (+ 1), (+ 1) . (+ 1) . (+ 1), ... ]`.

*Soluție.* .

```

1 compositions :: (a -> a) -> [a -> a]
2 compositions f = f : map (f .) (compositions f)

```

□

6. Scrieți în Haskell o funcție care întoarce `True` dacă un element căutat se află într-o listă, eventual **infinită**, folosind funcționala `foldr`. Este acest lucru posibil? Dacă da, definiți-o și explicați de ce funcționează. Dacă nu, justificați de ce nu se poate.

*Soluție.* .

```

1 exists :: Eq a => a -> [a] -> Bool
2 exists element = foldr (||) False . map (== element)

```

Este posibil, deoarece funcția trimisă lui `foldr` poate alege dacă să evaluateze sau nu acumulatorul, în baza evaluării lenșe. În cazul de față, acumulatorul este al doilea parametru al lui `(||)`, nemaifiind evaluat dacă primul este `True`.  $\square$

7. Sintetizați tipul următoarei funcții Haskell:

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

*Soluție.* .

```

1 f :: a -> b -> [c]
2 c = (a, b) = (b, a)
3 a = b
4 f :: a -> a -> [(a, a)]

```

$\square$

8. Să presupunem că numerele complexe sunt reprezentate în Haskell sub forma unor perechi numerice, e.g.  $2 + 3i \equiv (2, 3)$ . Instantiați clasa `Num` cu tipul menționat mai sus, supraîncărcând doar operatorul `+`. De exemplu,  $(2, 3) + (20, 30) = (22, 33)$ .

```

1 class Num a where
2     (+) :: a -> a -> a

```

*Soluție.* .

```

1 instance (Num a, Num b) => Num (a, b) where
2     (x1, y1) + (x2, y2) = (x1 + x2, y1 + y2)

```

$\square$

9. Transcrieți în logica cu predicate de ordinul I următoarea propoziție:

*Cine sapă groapa altuia, cade el în ea.*

*Soluție.* .

Utilizăm predicatele

- `sapa(agresor, groapa, victima)`
- `cade(persoana, groapa).`

$$\forall a. \forall g. (\exists v. sapa(a, g, v) \Rightarrow cade(a, g))$$

$\square$

10. Utilizați rezoluția propozițională pentru a demonstra că propoziția  $p \Leftrightarrow \neg p$  este contradicție. Rezolvările care nu utilizează rezoluția **nu** vor fi punctate!

*Soluție.* .

Etape:

- Rescriere:  $(p \Rightarrow \neg p) \wedge (\neg p \Rightarrow p)$
- Eliminarea implicațiilor:  $(\neg p \vee \neg p) \wedge (p \vee p)$
- Forma clauzală:  $\{\neg p\}, \{p\}$

- Rezoluție pe cele două clauze: {}

Obținerea clauzei vide indică prezența unei contradicții.  $\square$

11. De câte ori se satisfac următorul scop în Prolog și care sunt legăurile aferente?

```
append(X, Y, [1, 2, 3, 4]), member(_, X), member(L, Y), length(Y, L) .
```

*Soluție.* .

Datorită condiției `member(_, X)`,  $X$  conține cel puțin un element. Situații:

- $X = [1], Y = [2, 3, 4]$ 
  - $L = 2$ , `length([2, 3, 4], 2)`, eșec
  - $L = 3$ , `length([2, 3, 4], 3)`, ok
  - $L = 4$ , `length([2, 3, 4], 4)`, eșec
- $Y = [3, 4]$  sau  $Y = [4]$  sau  $Y = []$ 
  - 3 și 4 sunt mai mari ca 2, 1, respectiv 0, eșec.

O singură satisfacere.  $\square$

12. Scrieți un algoritm Markov care calculează cîtul împărțirii la 2 a unui număr natural. Numărul este reprezentat unar, în forma unei secvențe de simboluri 1, de lungime egală cu numărul. De exemplu, atât pentru numărul 3, reprezentat prin secvența 111, cât și pentru numărul 2, reprezentat prin secvența 11, banda va conține la sfârșitul execuției simbolul 1.

*Soluție.* .

În programul de mai jos,  $a$  este o variabilă de lucru.

```
1 Mod2()
2     a11 -> 1a
3     a1 -> a
4     a -> .
5     -> a
6 end
```

Exemplu: 11 -5-> a11 -2-> 1a -4-> 1.

Exemplu: 111 -5-> a111 -2-> 1a1 -3-> 1a -4-> 1.  $\square$

Precizări:

- Fiecare subiect are câte 10p. **Punctajul NU se acordă în absență justificării răspunsului!**
- Este suficientă rezolvarea a **10** subiecte din cele **12** pentru nota maximă.
- Punctajul suplimentar reprezintă **bonus**, ce poate compensa punctajul de pe parcurs.

1. Îmbogătiți λ-expresia  $(x \ x)$ , astfel încât expresia finală să conțină exact 2 apariții **legate** ale variabilei  $x$ .

*Soluție.* .

$(\lambda x.x \ x)$  sau  $(x \ \lambda x.x)$ . □

2. (20p!) Precizați cum decurg, pas cu pas, evaluările expresiilor Racket de mai jos, utilizând modelul de evaluare bazat pe **substituție** textuală. Există vreo diferență? Dacă da, în ce constă?

(a) (10p)

```
1  (and (> (+ 1 2) 5)
2      (> (+ 2 3) 5))
```

(b) (10p)

```
1  (define (f x y)
2      (and (> x 5)
3          (> y 5)))
4  (f (+ 1 2) (+ 2 3))
```

*Soluție.* .

(a)  $\begin{aligned} & (\text{and } (> (+ 1 2) 5) \quad (> (+ 2 3) 5)) \\ & \rightarrow (\text{and } \underline{(> 3 5)} \quad (> (+ 2 3) 5)) \\ & \rightarrow (\text{and } \#f \quad \underline{(> (+ 2 3) 5)}) \\ & \rightarrow \#f \end{aligned}$

(b)  $\begin{aligned} & (f \underline{(+ 1 2)} \quad (+ 2 3)) \\ & \rightarrow (f 3 \underline{(+ 2 3)}) \\ & \rightarrow (f 3 \underline{5}) \\ & \rightarrow (\text{and } \underline{(> 3 5)} \quad (> 5 5)) \\ & \rightarrow (\text{and } \#f \quad \underline{(> 5 5)}) \\ & \rightarrow \#f \end{aligned}$  □

3. Funcțiile Racket de mai jos determină **ultimul** element al unei liste nevide în două moduri diferite:

```
1  (define (last1 L)
2      (if (null? (cdr L))
3          (car L)
4          (last1 (cdr L))))
5
6  (define last2 (compose car reverse))
```

Scrieti un avantaj al folosirii lui `last1` în locul lui `last2`. Se presupune că `reverse` este implementată eficient.

*Soluție.* .

`last2` necesită construcția listei inversate, în timp ce `last1`, nu.  $\square$

4. În Haskell, funcționala `map` se poate extinde pentru a opera cu funcții parțiale, care întorc `Nothing` dacă nu sunt definite într-un anumit punct, în forma

```
1 mapMaybe :: (a -> Maybe b)  
2           -> [a] -> [b]
```

unde constructorul de tip `Maybe` este definit ca

```
1 data Maybe a = Nothing | Just a
```

Diferența față de funcționala `map` standard constă în posibilitatea funcționalei `mapMaybe` de a *renunța* la elementele pentru care funcția de aplicat întoarce `Nothing`. De exemplu, expresia de mai jos se va evalua la `[40, 60]`.

```
1 mapMaybe (\x -> if x == 5  
2               then Nothing  
3               else Just $ x * 10)  
4 [4, 5, 6]
```

Implementați funcționala `mapMaybe`, folosind **exclusiv funcționale**. Rezolvările care nu respectă condiția **NU** vor fi punctate!

*Soluție.* .

```
1 mapMaybe :: Eq b => (a -> Maybe b) -> [a] -> [b]  
2 mapMaybe f = map (\(Just x) -> x) . filter (/= Nothing) . map f
```

Constrângerea (`Eq b`) este revendicată de operatorul `(/=)`, fiind specifică acestei implementări.  $\square$

5. La ce se evaluează expresia Racket de mai jos? Justificați!

```
1 (let ([x 1])  
2   (lambda (x)  
3     (let ([x x])  
4       x)))
```

*Soluție.* .

Corpul construcției `let` este o funcție, care va constitui și valoarea expresiei. `let`-ul interior leagă o variabilă locală la parametrul funcției, și o întoarce. Valoarea va fi deci funcția identitate.  $\square$

6. Sintetizați tipul funcției Haskell de mai jos. Justificați!

$$f \ x \ y = [x \ y, \ y]$$

*Soluție.* .

```

1  f :: a -> b -> c
2  x :: d -> e
3  a = d -> e
4  c = [e]
5  b = d
6  b = e
7  f :: (e -> e) -> e -> [e]

```

□

7. Fie reprezentarea în Haskell a unei matrice pătratice sub forma unei liste de linii. Mai jos, sunt ilustrate o matrice și reprezentarea ei:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow [[1, 2], [3, 4]]$$

Supraîncărcați operatorul (+) pentru a realiza adunarea matricelor reprezentate ca mai sus. Pentru aceasta, instantiați clasa Num:

```

1  class Num a where
2      (+) :: a -> a -> a

```

*Hint:*

```

1  zipWith :: (a -> b -> c)
2      -> [a] -> [b] -> [c]

```

*Soluție.* .

```

1  {-# LANGUAGE FlexibleInstances #-}
2
3  instance Num a => Num [[a]] where
4      (+) = zipWith (zipWith (+))

```

□

8. Scrieți o propoziție din logica propozițională care să admită **exact** 42 de interpretări distințte. Justificați!

*Soluție.* .

Nu există o astfel de expresie, deoarece numărul de interpretări este  $2^n$ , unde  $n$  este numărul de propoziții simple.

□

9. Utilizați rezoluția propozițională pentru a demonstra că

$$\{a \vee b, a \Rightarrow c, b \Rightarrow c\} \models c.$$

*Soluție.* .

Adăugând negația concluziei,  $\neg c$ , la setul de ipoteze, și transformând în forma clauzală, se obțin clauzele:  $\{a, b\}$ ,  $\{\neg a, c\}$ ,  $\{\neg b, c\}$ ,  $\{\neg c\}$ . De aici, se poate obține cu ușurință clauza vidă.

□

10. Transcrieți în Prolog propoziția:

$$\neg \exists x. (p(x) \wedge \neg q(x)).$$

*Soluție.* .

Propoziția se poate scrie în forma  $\forall x.(\neg p(x) \vee q(x))$ , respectiv  $\forall x.(p(x) \Rightarrow q(x))$ . În Prolog, aceasta se poate reprezenta prin:

1  $q(X) :- p(X).$  □

11. Ce sir se va afla pe banda mașinii Markov după execuția algoritmului de mai jos, presupunând că sirul inițial este abcbc?

1 abc -> ca

2 cc ->

3 -> .

*Soluție.* .

abcbc -1-> cabc -1-> cca -2-> a □

Precizări:

- Fiecare subiect are câte 10p. **Punctajul NU se acordă în absență justificării răspunsului!**
- Este suficientă rezolvarea a **10** subiecte din cele **12** pentru nota maximă.
- Punctajul suplimentar reprezintă **bonus**, ce poate compensa punctajul de pe parcurs.

1. Fie următoarea reprezentare a numerelor naturale în calculul lambda:  $0 \equiv \lambda x.x$ ,  $1 \equiv \lambda x.\lambda x.x$ ,  $2 \equiv \lambda x.\lambda x.\lambda x.x$  etc. Scrieți definiția funcției de incrementare, *inc*, astfel încât  $(inc\ 0) \rightarrow 1$ ,  $(inc\ 1) \rightarrow 2$  etc.

*Soluție.* .

$$inc \equiv \lambda n.\lambda x.n.$$

□

2. Fie următoarea funcție în Racket:

```
1  (define (f x)
2    (lambda (y)
3      (+ (* x 3) y)))
```

Cum decurge evaluarea aplicației  $(f\ (+\ 1\ 2))$ , utilizând modelele bazate pe

- (a) (5p) substituție textuală
- (b) (5p) evaluare contextuală?

*Soluție.* .

Ambele situații presupun mai întâi evaluarea parametrului la 3.

- (a)  $\lambda y. (+ (* 3 3) y)$
- (b)  $\langle \lambda y. (+ (* x 3) y); \{x \leftarrow 3\} \rangle$

□

3. Fie următoarea funcție în Racket:

```
1  (define (f x)
2    (lambda (y)
3      (if (eq? y 'halt)
4        x
5        (f (+ x y)))))
```

- (a) (5p) Ce face *f*?
- (b) (5p) Dați un exemplu de expresie care surprinde utilitatea lui *f*.

*Soluție.* .

- (a) *f* calculează suma parametrilor primiți până la întâlnirea simbolului 'halt.
- (b)  $((((f\ 1)\ 2)\ 3)\ 'halt) \rightarrow 6$

□

4. Fie următoarea reprezentare în Haskell a unui arbore binar:

```

1 data Tree a
2     = Empty
3     | Node (Tree a) a (Tree a)

```

Implementați pentru tipul `Tree a` o funcțională analoagă lui `foldr`, care să acumuleze toate cheile din arbore în ordinea subarbore drept – rădăcină – subarbore stâng, pornind de la definițiile de mai jos:

```

1 foldr :: (a -> b -> b) -> b -> [a] -> b
2 foldr f acc [] = acc
3 foldr f acc (h : t) = f h (foldr f acc t)
4
5 foldTree :: (a -> b -> b) -> b -> Tree a -> b
6 ...

```

*Hint:* O listă poate fi privită ca un arbore în care niciun nod nu are subarbore stâng.

*Soluție.* .

```

1 foldT _ acc Empty = acc
2 foldT f acc (Node left key right) =
3     foldT f (f key (foldT f acc right)) left

```

□

5. Care sunt valorile expresiilor din liniile 7 și 10 în urma execuției următoarei secvențe în Racket (presupunând utilizarea unui limbaj care permite `define`-uri multiple)?

```

1 (define f
2   (let ([x 1])
3     (lambda (y)
4       (+ x y))))
5
6 (define x 0)
7 (let ([x 10]) (f 0))
8
9 (define x 10)
10 (let ([x 0]) (f 0))

```

*Soluție.* .

Ambele valori sunt 1, pentru că apariția lui `x` din linia 4 este obținută static pe baza definiției din linia 2.

□

6. Fie următoarea definiție și aplicare în linia de comandă a unei funcții în Haskell:

```

1 f x y = x + x
2 > f (1 + 2) (3 + 4)

```

Rescrieți cele două expresii în Racket, astfel încât să obțineți aceleași efecte ca în Haskell, în privința modalității de evaluare.

*Soluție.* .

```

1 (define (f x y)
2   (+ (force x) (force y)))
3
4 > (f (delay (+ 1 2)) (delay (+ 3 4)))

```

□

7. Sintetizați tipul următoarei funcții în Haskell:

```
1 f x y z = foldr (.) id [x, y, z]
```

Tipul operatorului de compunere este

```
1 (.) :: (d -> e) -> (c -> d) -> c -> e
```

*Soluție.* .

```

1 f :: g -> h -> i -> j
2 foldr :: (a -> b -> b) -> b -> [a] -> b
3 a = d -> e -- foldr + (.)
4 b = c -> d = c -> e -- foldr + (.)
5 b = k -> k -- foldr + id
6 c = d = e = k
7 g = h = i = a -- foldr + lista
8 f :: (k -> k) -> (k -> k) -> (k -> k) -> k -> k

```

□

8. Supraîncărcați în Haskell operatorul de adunare pentru funcții unare. Într-un punct oarecare, valoarea funcției rezultate se obține prin însumarea valorilor funcțiilor în punctul respectiv.

*Soluție.* .

```

1 instance Num b => Num (a -> b) where
2   (f + g) x = f x + g x
3   -- sau: f + g = \x -> f x + g x

```

□

9. Fie propozițiile din logica cu predicate de ordinul I:

$$\forall x. \exists y. (x \leq y) \quad (1)$$

$$\exists y. \forall x. (x \leq y) \quad (2)$$

- (a) (5p) Sub o interpretare în care domeniul este reprezentat de multimea numerelor naturale, care dintre cele două propoziții este adevărată?
- (b) (5p) Cum trebuie modificat domeniul pentru ca ambele propoziții să fie adevărate?

*Soluție.* .

- (a) Doar propoziția (1) este adevărată, deoarece orice număr natural este mai mic sau egal cu sine ( $y = x$ ), însă nu putem găsi un cel mai mare număr natural, aşa cum pretinde propoziția (2).
- (b) Domeniul trebuie să fie finit. □

10. Utilizând metoda rezoluției, demonstrați că ( $a$  este o constantă):

$$\{\forall x.P(x) \Rightarrow Q(x), P(a)\} \models \exists x.Q(x).$$

*Soluție.* .

Utilizăm reducerea la absurd, prin care adăugăm negația concluziei la mulțimea de propoziții:  $\neg\exists x.Q(x) \equiv \forall x.\neg Q(x)$ . Astfel, în urma aplicării procedurii de transformare în forma normală conjunctivă, care presupune și redenumirea unor variabile, se obțin clauzele  $\{\neg P(x), Q(x)\}, \{P(a)\}, \{\neg Q(y)\}$ . Aplicând rezoluția în raport cu substituția  $\{y \leftarrow x, x \leftarrow a\}$ , se obține clauza vidă.  $\square$

11. Transcrieți ca o **unică** propoziție logică următorul program Prolog. Punctajul **NU** se acordă pentru mai mult de o propoziție echivalentă.

```
1 c :- a.    c :- b.  
2 d :- a.    d :- b.
```

*Soluție.* .

$$a \vee b \Rightarrow c \wedge d.$$

$\square$

12. Scrieți un algoritm Markov care să elimine toate copiile **adiacente** ale unui simbol de pe bandă, păstrând un singur exemplar. Spre exemplu, dacă sirul inițial este 122233112, se va obține în final 12312.

*Soluție.* .

```
1 Clean()  
2     gg -> g  
3     -> .  
4 end
```

$\square$

Precizări:

- Fiecare subiect are câte 10p. **Punctajul NU se acordă în absență justificării răspunsului!**
- Este suficientă rezolvarea a **10** subiecte din cele 12 pentru nota maximă.
- Punctajul suplimentar reprezintă **bonus**, ce poate compensa punctajul de pe parcurs.

1. Care este rezultatul aplicației  $(\lambda x.(\lambda x.x\ x)\ a)$ , unde

$$a \equiv \lambda y.\lambda z.\lambda s.\lambda w.\lambda t.(((y\ t)\ (z\ w))\ s)?$$

*Hint:* Imagineați-vă cum decurge evaluarea conform modelului bazat pe substituție textuală, învățat la Racket.

*Soluție.* .

$$(\lambda x.(\lambda x.x\ x)\ a) \rightarrow (\lambda x.x\ a) \rightarrow a.$$

□

2. Se urmărește implementarea în Racket a funcției de concatenare a listelor, append, utilizând recursivitatea **pe coadă**. Pentru aceasta, fie funcția **f** de mai jos:

```
1 (define (f X Y)
2   (if (null? X) Y
3       (f (cdr X) (cons (car X) Y))))
```

(a) (2p) Implementați funcția de inversare, reverse, utilizând **exclusiv f**.

(b) (8p) Implementați funcția de concatenare, append, utilizând **exclusiv f și reverse**.

*Soluție.* .

```
1 (define (rev A)
2   (f A '()))
3
4 (define (appl A B)
5   (f (rev A) B))
6 ; sau
7 (define (app2 A B)
8   (rev (f B (rev A))))
```

□

3. Fie următorul tip de arbore oarecare nevid în Haskell:

```
1 data Tree a = Node
2   { key      :: a
3   , children :: [Tree a]
4   } deriving (Eq, Show)
```

Definiți pentru tipul **Tree a** o funcțională analoagă lui **filter**, care să păstreze dintr-un arbore doar nodurile a căror cheie îndeplinește condiția precizată. Pentru simplitate, rădăcina **NU** va fi verificată. Funcția va avea tipul de mai jos și va utiliza **cel puțin o funcțională standard**:

```
1 filterTree :: (a -> Bool) -> Tree a -> Tree a
```

*Soluție.* .

```
1 filterTree f (Node k ch) =  
2     Node k $ map (filterTree f) $ filter (f . key) ch
```

□

4. Care este valoarea ultimei expresii din codul de mai jos?

```
1 (define (add x)  
2   (lambda (y)  
3     (+ x y)))  
4  
5 (define x 10)  
6  
7 (let ([x 20])  
8   (map (add x) '(1 2 3)))
```

*Soluție.* .

'(21 22 23), întrucât valoarea lui x este cea din let.

□

5. Care sunt valorile expresiilor din liniile 6, 7, 11 și 12, din următorul cod Racket?

```
1 (define x 0)  
2  
3 (define closure (lambda () x))  
4 (define promise (delay x))  
5  
6 (closure)  
7 (force promise)  
8  
9 (define x 1)  
10 (closure)  
11 (force promise)
```

*Soluție.* .

0, 0, 1, 0. Linia 11 utilizează valoarea calculată în linia 6, la prima forțare (0), în timp ce linia 12 presupune reevaluarea corpului x, care acum este 1.

□

6. Definiți în Haskell fluxul prefixelor unui alt flux. Spre exemplu, avem:

```
1 > take 5 $ prefixes [0..]  
2 [[], [0], [0,1], [0,1,2], [0,1,2,3]]
```

*Soluție.* .

```
1 prefixes :: [a] -> [[a]]  
2 prefixes s = out  
3   where  
4       out = [] : zipWith f out s  
5           f prefix x = prefix ++ [x]
```

□

7. Sintetizați tipul următoarei funcții în Haskell:

```
1 f g (x, y, z) = map g [x, y, z]
```

*Soluție.* .

```
1 f :: (a -> b) -> (a, a, a) -> [b]
```

x, y și z trebuie să aibă același timp, deoarece sfârșesc în aceeași listă.  $\square$

8. Supraîncărcați în Haskell operatorul de adunare pentru tipul predefinit

```
1 data Maybe a = Just a | Nothing
```

Ori de câte ori un operand este Nothing, rezultatul va fi Nothing.

*Soluție.* .

```
1 instance Num a => Num (Maybe a) where
2     Just x + Just y = Just $ x + y
3     _ + _ = Nothing
```

$\square$

9. Care este diferența dintre următoarele expresii logice?

$$\forall x.(P(x) \Rightarrow Q(x)) \quad (1)$$

$$\forall x.P(x) \Rightarrow \forall x.Q(x) \quad (2)$$

*Soluție.* .

Propoziția (1) afiră că, ori de câte ori proprietatea  $P$  este îndeplinită de către un obiect, trebuie să fie satisfăcută și proprietatea  $Q$  de același obiect. Propoziția (2) afiră că, în condițiile în care proprietatea  $P$  este satisfăcută de *toate* obiectele, același lucru trebuie să se întâmple și cu proprietatea  $Q$ .  $\square$

10. Utilizând metoda rezoluției, demonstrați că:

$$\{\exists x.(P(x) \wedge Q(x))\} \models \exists x.P(x).$$

*Soluție.* .

Construim conjuncția între ipoteză și negația concluziei, și o transformăm în FNC:

$$\begin{aligned} & \exists x.(P(x) \wedge Q(x)) \wedge \neg \exists x.P(x) \\ & \exists x.(P(x) \wedge Q(x)) \wedge \forall x.\neg P(x) \\ & \exists x.\forall y.(P(x) \wedge Q(x) \wedge \neg P(y)) \\ & P(c_x) \wedge Q(c_x) \wedge \neg P(y) \\ & \{P(c_x)\}, \{Q(c_x)\}, \{\neg P(y)\} \end{aligned}$$

Aplicând rezoluția pentru clauzele  $\{P(c_x)\}$  și  $\{\neg P(y)\}$ , utilizând substituția  $\{y \leftarrow c_x\}$ , obținem clauza vidă.  $\square$

11. Scrieți un predicat Prolog care numără toate aparițiile unui element dintr-o listă. Descrieți semnificația parametrilor.

*Soluție.* .

```
1 count (_, [], 0).
2 count (X, [H|T], N) :- count (X, T, N1), (X == H, N is N1 + 1, ! ; N is
N1).
```

□

12. Scrieți un algoritm Markov care, pentru orice secvență alternantă de trei simboluri 0 sau 1, să înlocuiască simbolul din mijloc cu cel din extremități. Spre exemplu, dacă sirul inițial este 010101, se va obține în final 000111.

*Soluție.* .

```
1 Uniform()
2     g1g2g1 -> g1g1g1
3     -> .
4 end
```

□

Precizări:

- Primele 9 subiecte au fiecare câte 10p. Cele 3 subpunkte ale problemei au fiecare câte 10p. **Punctajul NU se acordă în absență justificării răspunsului!**
- Este suficientă rezolvarea a **10** itemi din cei 12 pentru nota maximă.
- Punctajul suplimentar reprezintă **bonus**, ce poate compensa punctajul de pe parcurs.

1. Fie definiția și aplicația funcției Racket de mai jos. Descrieți **pas cu pas** cum decurge evaluarea aplicației, utilizând modelul bazat pe substituție textuală.

```
1  (define (f x y)
2    (lambda (z)
3      (if (< x 10) (+ x y) z)))
4
5  > ((f (+ 0 1) (+ 2 3)) (+ 4 5))
```

*Soluție.* .

```
1  ((f (+ 0 1) (+ 2 3)) (+ 4 5))
2  → ((f 1 5) (+ 4 5))
3  → ((lambda (z) (if (< 1 10) (+ 1 5) z)) (+ 4 5))
4  → ((lambda (z) (if (< 1 10) (+ 1 5) z)) 9)
5  → (if (< 1 10) (+ 1 5) 9)
6  → (if true (+ 1 5) 9)
7  → (+ 1 5)
8  → 6
```

□

2. (V. sub. 1) Fie definiția și aplicația funcției Haskell de mai jos. Descrieți **pas cu pas** cum decurge evaluarea aplicației, utilizând modelul bazat pe substituție textuală.

```
1  f x y z = if x < 10 then x + y else z
2
3  > f (0 + 1) (2 + 3) (4 + 5)
```

*Soluție.* .

```
1  f (0 + 1) (2 + 3) (4 + 5)
2  → if (0 + 1) < 10 then (0 + 1) + (2 + 3) else (4 + 5)
3  → if 1 < 10 then 1 + (2 + 3) else (4 + 5)
4  → if True then 1 + (2 + 3) else (4 + 5)
5  → 1 + (2 + 3)
6  → 1 + 5
7  → 6
```

□

3. Fie următoarea funcție Racket:

```
1  (define (f x)
2    (if (<= x 0) 0
3        (- (f (- x 1)) 1))) ; (f (- (f (- x 1)) 1)))
```

- (a) Ce fel de recursivitate utilizează funcția?

- (b) Ce fel de recursivitate utilizează funcția, dacă înlocuim ultima linie cu expresia comentată?

*Soluție.* .

- (a) Pe stivă, întrucât se mai realizează operații după evaluarea aplicației recursive.  
 (b) Tot pe stivă, datorită aplicației recursive interne.  $\square$

4. Ce afișează programul Racket de mai jos?

```

1  (define (x x)
2    (let ([x x])
3      (x x)))
4
5  (x 0)
```

*Soluție.* .

Eroare, deoarece aparițiile lui  $x$  din linia 3 se referă la parametrul funcției, 0, care nu poate fi aplicat ca o funcție.  $\square$

5. Sintetizați tipul următoarei funcții Haskell:

```
1  f u = map . u
```

Tipul operatorului de compunere este

```
1  (.) :: (b -> c) -> (a -> b) -> a -> c
```

*Soluție.* .

```

1  f :: d -> e
2  u :: d = a -> b
3  map :: (g -> h) -> [g] -> [h]
4  b = g -> h
5  c = [g] -> [h]
6  e = a -> c
7  f :: (a -> g -> h) -> a -> [g] -> [h]
```

$\square$

6. Supraîncărcați în Haskell operatorul ( $<=$ ) pe liste, astfel încât o listă este mai mică sau egală decât alta, dacă același lucru se poate afirma și despre sumele elementelor lor. Exemplu:  $[1, 2, 10] <= [0, 1, 100, 3]$ , întrucât  $13 <= 104$ .

*Soluție.* .

```

1  instance (Num a, Ord a) => Ord [a] where
2    xs <= ys = sum xs <= sum ys
```

$\square$

7. Câte interpretări satisfac propozițiile  $\alpha$ , respectiv  $\beta$ , în logica propozițională, unde  $p_1, \dots, p_{10}$  sunt propoziții simple?

$$\begin{aligned}\alpha &= p_1 \vee \dots \vee p_{10} \\ \beta &= (p_1 \vee \dots \vee p_5) \wedge (p_6 \vee \dots \vee p_{10})\end{aligned}$$

*Soluție.* .

Pentru  $n$  propoziții simple, există  $2^n$  interpretări distințe.

Toate interpretările satisfac  $\alpha$ , cu excepția celei care asociază Fals fiecărei propoziții simple. Numărul final este  $2^{10} - 1$ .

Similar se poate judeca și în cazul celor două paranteze din  $\beta$ . Numărul final este  $(2^5 - 1)^2$ .

□

8. Utilizând metoda rezoluției, demonstrați că:

$$\{\forall x.P(x) \vee \forall x.Q(x)\} \models \forall x.(P(x) \vee Q(x)).$$

*Soluție.* .

Pentru claritate, redenumim, variabilele, obținând:

$$\{\forall x.P(x) \vee \forall y.Q(y)\} \models \forall z.(P(z) \vee Q(z)).$$

Utilizăm reducerea la absurd, conform căreia adăugăm negația concluziei la mulțimea de propoziții:  $\neg\forall z.(P(z) \vee Q(z)) \equiv \exists z.(\neg P(z) \wedge \neg Q(z))$ . Astfel, în urma aplicării procedurii de transformare în forma normală conjunctivă, care presupune și înlocuirea variabilei cu antificate existențial  $z$  cu constanta  $c_z$ , se obțin clauzele  $\{P(x), Q(y)\}, \{\neg P(c_z)\}, \{\neg Q(c_z)\}$ . Aplicând rezoluția în raport cu substituția  $\{x \leftarrow c_z, y \leftarrow c_z\}$ , se obține clauza vidă. □

9. Fie următorul program Prolog:

```
1  a(1).  a(2).
2  b(1).  b(2).
3
4  c(X,  Y) :- a(X),  b(Y).
5  c(3,  4).
6  c(4,  3).
```

Unde ar trebui plasat operatorul `cut`, astfel încât scopul următor să fie satisfăcut?

```
1  ?- findall(_,  c(X,  Y),  L),  length(L,  4).
```

*Soluție.* .

```
1  c(X,  Y) :- !,  a(X),  b(Y).
```

Astfel, se generează doar cele patru combinații între `a` și `b`, renunțându-se la definițiile din liniile 5 și 6. □

10. PROBLEMA. Puteți utiliza oricare dintre cele două limbaje **funcționale** studiate.

- Definiți reprezentarea unui arbore oarecare nevid, în care un nod poate avea oricărți copii.
- Definiți o funcție care calculează înălțimea arborelui. Înălțimea unei frunze este 0. Utilizați cel puțin o funcțională.
- Definiți o funcție care calculează numărul de frunze din arbore. Utilizați cel puțin o funcțională.

*Soluție.* .

Exemplificăm soluția în Haskell.

```
1 data Tree a = Node a [Tree a]
2     deriving (Eq, Show)
3
4 height :: Tree a -> Int
5 height (Node _ []) = 0
6 height (Node _ children) = 1 + maximum (map height children)
7
8 leaves :: Tree a -> Int
9 leaves (Node _ []) = 1
10 leaves (Node _ children) = sum (map leaves children)
```

□

Precizări:

- Fiecare subiect are câte 10p.
- Este suficientă rezolvarea a 10 subiecte din cele 12 pentru nota maximă.
- Punctajul suplimentar reprezintă bonus, ce poate compensa punctajul de pe parcurs.

1. Care este rezultatul evaluării aplicației  $(x \ \lambda x.x)$ , în absența oricăror informații despre variabila liberă  $x$ ? Justificați!

*Soluție.* .

Evaluarea se blochează deoarece membrul stâng al aplicației nu poate fi adus la forma funcțională.  $\square$

2. Implementați în Haskell operatorul `(++)`, întrebuițând una din funcțiile **fold\*** și **fără** a utiliza recursivitate explicită. Soluțiile care nu respectă cele două constrângeri **nu** vor fi punctate!

```
foldl :: (a -> b -> a)
          -> a -> [b] -> a
foldr :: (a -> b -> b)
          -> b -> [a] -> b
```

*Soluție.* .

```
1  append xs ys = foldr (:) ys xs
```

$\square$

3. La ce se evaluează ultima expresie a programului Scheme de mai jos? Justificați!

```
1  (define f (lambda () x))
2  (let ([x 1]) (f))
```

*Soluție.* .

Evaluarea generează eroare, deoarece variabila `x` din corpul funcției `f` nu este găsită. Din moment ce construcția `let` realizează legare statică, domeniul de vizibilitate a lui `x`, definit aici, nu include funcția `f`.  $\square$

4. Fie funcția Scheme de mai jos:

```
1  (define f
2    (lambda (x y)
3      (if (< x 5)
4        (+ x x)
5        (+ y y))))
```

Precizați cum decurge, pas cu pas, evaluarea expresiei `(f (+ 1 2) (+ 2 3))`, utilizând modelul de evaluare bazat pe **substituție** textuală.

*Soluție.* .

$$\begin{aligned} & (\text{f } \underline{(+ \ 1 \ 2)} \quad (+ \ 2 \ 3)) \\ \rightarrow & (\text{f } 3 \ \underline{(+ \ 2 \ 3)}) \\ \rightarrow & \underline{(\text{f } 3 \ 5)} \\ \rightarrow & (\text{if } \underline{(< \ 3 \ 5)} \quad (+ \ 3 \ 3) \quad (+ \ 5 \ 5)) \\ \rightarrow & \underline{(\text{if } \#t \ (+ \ 3 \ 3) \quad (+ \ 5 \ 5))} \\ \rightarrow & \underline{(+ \ 3 \ 3)} \\ \rightarrow & 6 \end{aligned}$$

□

5. Fie funcția Haskell de mai jos:

```
1 f :: Int -> Int -> Int
2 f 2013 _ = 2013
3 f x     y = x + y
```

Ce puteți spune despre terminarea evaluării următoarei expresii Haskell?

**foldr** f 0 [0 ..]

*Soluție.* .

Se termină, deoarece, conform definiției lui **f**, dacă elementul curent al listei este 2013, aplicația recursivă a lui **foldr** asupra restului listei nu mai este evaluată, ca urmare a strategiei leneșe. □

6. Definiți în Haskell fluxul rezultat prin aplicarea succesivă a unei funcții cu tipul  $a \rightarrow a$  asupra unei expresii initiale de tipul  $a$ . De exemplu, pentru funcția  $(+ \ 1)$  și valoarea initială 0, se obține fluxul  $[0, 1, 2, 3, \dots]$ .

*Soluție.* .

```
1 apply f x = x : apply f (f x)
```

□

7. Sintetizați tipul operatorului  $(.)$ , având definiția:

$$(.) \ f \ g \ x = f (g \ x)$$

*Soluție.* .

```
1 (.) :: a -> b -> c -> d
2 f :: e -> h
3 g :: i -> j
4 a = e -> h
5 b = i -> j
6 c = i
7 e = j
8 d = h
9 (.) :: (j -> h) -> (i -> j) -> i -> h
```

□

8. Pentru tipul de date Natural, definit mai jos, supraîncărcați explicit (fără a utiliza deriving) funcția standard de verificare a egalității.

```

1  data Natural
2      = Zero
3      | Succ Natural

```

*Soluție.* .

```

1  instance Eq Natural where
2      Zero      == Zero      = True
3      Zero      == (Succ _) = False
4      (Succ _) == Zero      = False
5      (Succ m) == (Succ n) = m == n

```

□

9. Utilizați rezoluția propozițională pentru a demonstra că propoziția  $p$  este o consecință logică a propoziției  $p \wedge q$ .

*Soluție.* .

Forma clauzală a mulțimii de propoziții, formată din ipoteză și negația concluziei, este  $\{p\}, \{q\}, \{\neg p\}$ . Aplicând rezoluția asupra clauzelor  $\{p\}$  și  $\{\neg p\}$ , se obține clauza vidă.

□

10. Definiți în Prolog predicatul xor/2, care să fie satisfăcut dacă **numai unul** din cei doi parametri este satisfăcut. **Nu** puteți folosi predicatul not sau operatorul \+.

*Soluție.* .

```

1  xor(X, Y) :- X, Y, !, fail.
2  xor(X, _) :- X, !.
3  xor(_, Y) :- Y.

```

□

11. Scrieți un algoritm Markov, care calculează predecesorul unui număr natural nenul. Numerele sunt reprezentate unar, în forma unei secvențe de simboluri 1, de lungime egală cu numărul. De exemplu, pentru numărul 3, reprezentat prin secvența 111, banda va conține la sfârșitul execuției secvența 11.

*Soluție.* .

În programul de mai jos, a este o variabilă de lucru.

```

1  Predecesor()
2      a -> .
3      1 -> a
4  end

```

Exemplu: 111 -3-> a11 -2-> 11.

□

12. Dați trei exemple de fapte de lungimi diferite, care se potrivesc cu *pattern*-ul CLIPS (f \$? \$?).

*Soluție.* .

(f), (f 1), (f 1 2)

□

Precizări:

- Fiecare subiect are câte 10p.
- Este suficientă rezolvarea a 10 subiecte din cele 12 pentru nota maximă.
- Punctajul suplimentar reprezintă bonus, ce poate compensa punctajul de pe parcurs.

1. Care este numărul **minim** de  $\alpha$ -conversii ce trebuie realizate pentru evaluarea corectă a expresiei  $(\lambda x.\lambda x.x \ (\lambda x.x))$ ? Justificați!

*Soluție.* .

Niciuna, deoarece parametrul actual,  $\lambda x.x$ , nu conține variabile libere. □

2. Implementați în Haskell funcția `length`, întrebuițând una din funcționalele **fold\*** și **fără** a utiliza recursivitate explicită. Soluțiile care nu respectă cele două constrângeri **nu vor fi punctate!**

```
foldl :: (a -> b -> a)
          -> a -> [b] -> a
foldr :: (a -> b -> b)
          -> b -> [a] -> b
```

*Soluție.* .

```
1 len = foldr (\_ acc -> acc + 1) 0
```

□

3. La ce se evaluează ultima expresie a programului Scheme de mai jos? Justificați!

```
1 (define x 1)
2 (define x (lambda () x))
3 (x)
```

*Soluție.* .

#<procedure:x>, deoarece `define` leagă dinamic variabilele, și se ia în considerare ultima legare a lui `x`, la funcția însăși. □

4. În ce condiții expresia Scheme `(eq? (force x) (force x))` s-ar putea evalua la #f? (5p) Dați o posibilă definiție a lui `x`. (5p)

*Soluție.* .

În condițiile în care forțarea promisiunii `x` produce efecte laterale explicite. Exemplu:

```
1 (define x (delay (set! x 1) 0))
```

□

5. Fie funcția Haskell de mai jos:

```
1 f x y = if x < 5
2           then x + x
3           else y + y
```

Precizați cum decurge, pas cu pas, evaluarea expresiei  $f(1 + 2)(2 + 3)$ , utilizând modelul de evaluare bazat pe **substituție** textuală.

*Soluție.* .

```
f (1 + 2) (2 + 3)
→ if (1 + 2) < 5 then (1 + 2) + (1 + 2) else (2 + 3) + (2 + 3)
→ if 3 < 5 then 3 + 3 else (2 + 3) + (2 + 3)
→ if True then 3 + 3 else (2 + 3) + (2 + 3)
→ 3 + 3
→ 6
```

De remarcat că, o dată ce  $(1 + 2)$  este evaluat, în vederea comparației cu 5, valoarea sa rămâne salvată și în cadrul expresiei  $3 + 3$ .  $\square$

6. Definiți în Haskell fluxul prefixelor unui alt flux, primit ca parametru. De exemplu, pentru fluxul  $[0, 1, 2, \dots]$ , se obține fluxul  $[], [0], [0, 1], [0, 1, 2], \dots$ .

*Soluție.* .

```
1 prefixes (x : xs) = [] : (map (x :) (prefixes xs))
```

$\square$

7. Sintetizați tipul operatorului  $(\$)$ , având definiția:

$$(\$) f x = f x$$

*Soluție.* .

```
1 ($) :: a -> b -> c
2 f :: d -> e
3 a = d -> e
4 b = d
5 c = e
6 ($) :: (d -> e) -> d -> e
```

$\square$

8. Pentru tipul de date Natural, definit mai jos, supraîncărcați funcția standard de reprezentare sub formă de sir de caractere, utilizând valoarea numerică aferentă. De exemplu, valorii Succ (Succ Zero) îi va corespunde sirul "2".

```
1 data Natural
2     = Zero
3     | Succ Natural
```

*Soluție.* .

```
1 instance Show Natural where
2     show = show . value
3         where
4             value Zero      = 0
5                 value (Succ n) = 1 + (value n)
```

□

9. Fie propoziția de mai jos, din logica propozițională, toate propozițiile din definiție fiind simple. Câte interpretări distințe admite ea? Justificați!

$$\alpha = ((p \vee q \vee r) \wedge (s \vee \neg r) \wedge (\neg s \vee q)) \Leftrightarrow (t \vee u)$$

*Soluție.* .

Numărul de propoziții simple distințe din definiție este 6, astfel că numărul de interpretări este  $2^6 = 64$ . □

10. Transcrieți în Prolog propoziția  $\forall x. \neg \text{rotund}(x) \Rightarrow \neg \text{cerc}(x)$ .

*Soluție.* .

Propoziția este echivalentă cu  $\forall x. \text{cerc}(x) \Rightarrow \text{rotund}(x)$ .

```
1 rotund(X) :- cerc(X).
```

□

11. Scrieți un algoritm Markov, care calculează restul împărțirii la 2 a unui număr natural. Numerele sunt reprezentate unar, în forma unei secvențe de simboluri 1, de lungime egală cu numărul. De exemplu, pentru numărul 3, reprezentat prin secvența 111, banda va conține la sfârșitul execuției simbolul 1. Pentru numărul 2, reprezentat prin secvența 11, banda va ajunge vidă.

*Soluție.* .

În programul de mai jos, a este o variabilă de lucru.

```
1 Mod2()
2      11 ->
3      -> .
4 end
```

Exemplu: 11111 -2-> 111 -2-> 1 -3-> 1. □

12. Dați trei exemple de fapte de lungimi diferite, care se potrivesc cu *pattern*-ul CLIPS (`f ? 5 $?`).

*Soluție.* .

(f 0 5), (f 0 5 1), (f 0 5 1 2) □

# Examen PP varianta A — NOT EXAM MODE

31.05.2019

ATENȚIE: Aveți 2 ore · 1-9: 10p; 10: 30p · 100p pentru nota maximă · **Justificați** răspunsurile!

1. Reduceți expresia lambda  $E = (\lambda x.(x (\lambda y.z x)) \lambda x.x)$

*Soluție:*

$$\rightarrow (\lambda x.(x z) \lambda x.x) \rightarrow (\lambda x.x z) \rightarrow z$$

2. Se dă următorul cod Racket:

```
(define computation (delay (+ 5 5)))
(* 5 5)
(define (f x) (cons x (force computation)))
(map f '(1 2 3 4))
```

- (a) De câte ori se realizează adunarea?  
(b) Prima evaluare a adunării se realizează înainte sau după înmulțire?  
(c) Rescrieți codul pentru `computation` și pentru `f` folosind încideri funcționale în loc de promisiuni și răspundeți din nou la întrebările de la (a) și (b).

*Soluție:*

- (a) o singură dată, la prima evaluare a lui `computation`.  
(b) după înmulțire, atunci când se apelează prima oară (`force computation`)  
(c) 

```
(define computation (λ () (+ 5 5)))
(* 5 5)
(define (f x) (cons x (computation)))
(map f '(1 2 3 4))
```

acum se apelează de 4 ori, la fiecare evaluare a lui `computation`; dar prima dată tot după înmulțire.

3. Date fiind două liste de numere  $L_1$  și  $L_2$ , scrieți în Racket codul care produce o listă de perechi  $(x . n)$ , unde  $x$  este un element din  $L_1$ , iar  $n$  este numărul de apariții ale lui  $x$  în  $L_2$ . E.g. pentru  $L_1 = (1 4 5 3)$  și  $L_2 = (1 3 2 4 1 5 3 9)$  rezultatul este  $((1 . 2) (4 . 1) (5 . 1) (3 . 2))$ . Nu folosiți recursivitate explicită.

*Soluție:*

```
(map (lambda (x) (cons x (length (filter ((curry equal?) x) L)))) '(1 4 5 3))
sau
(map (lambda (x) (cons x (length (filter (lambda (y) (equal? x y)) L)))) '(1 4 5 3))
```

4. Sintetizați tipul următoarei funcții în Haskell:  $f x y = x y (y x)$

*Soluție:*

$x :: a \rightarrow b \rightarrow c$

$y :: a$ , dar și  $y :: (a \rightarrow b \rightarrow c) \rightarrow b \Rightarrow a = (a \rightarrow b \rightarrow c) \rightarrow b$  Eroare, deoarece  $a$  nu poate uniifica cu o expresie de tip care îl conține strict pe  $a$ .

5. (a) Câți pași de concatenare sunt realizați pentru evaluarea expresiei Racket

`(car (append '(1 2) '(3 4)))` ?

- (b) Dar pentru expresia Haskell `head $ [1, 2] ++ [3, 4]` ?

*Soluție:*

- (a) Se concatenează întregime listele, deci doi pași.

- (b) Este suficient un singur pas pentru ca `head` să întoarcă primul element.

6. Evidențiați o posibilă instantă a clasei Haskell de mai jos:

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

*Soluție:*

```
instance MyClass [] where
  f = head
```

7. Transformați propoziția „Nu tot ce zboară se mănâncă.” în logică cu predicate de ordinul întâi.

*Soluție:*

$\exists x. zboara(x) \wedge \neg se\_mananca(x)$  – există și lucruri care zboară și nu se mănâncă sau  
 $\neg(\forall x. zboara(x) \Rightarrow se\_mananca(x))$  – nu este adevărat că orice care zboară automat se și mănâncă

8. Se dă programul Prolog:

```
p(R, S) :- member(X, R),
           findall(Y, (member(Y, R), Y \= X), T), !, q(X, T, S).
q(X, A, [X|A]). q(X, [A|B], [A|C]) :- q(X, B, C).
```

Dacă predicatul **p** primește în primul argument o listă, la ce valori leagă al doilea argument? Câte soluții are interogarea **p([1, 2, 3, 4], S)**?

*Soluție:*

Ia primul element (și elimină duplicatele lui) și îl pune pe diverse poziții ale listei, inclusiv pe prima. Patru soluții: **[1, 2, 3, 4]**, **[2, 1, 3, 4]**, **[2, 3, 1, 4]**, **[2, 3, 4, 1]**

9. Se dau următoarele relații genealogice prin predicatul **c(Parinte, Copil)**. Implementați predicatul **frati(X, F)**, care leagă **F** la lista de frați ai lui **X** (dacă există). De exemplu, pentru definițiile de mai jos, interogarea **frati(herodot, F)** leagă **F** la **[faramir, george]**.

```
c(alex, celia). c(alex, delia). c(alex, marcel).
c(barbra, celia). c(barbra, delia). c(barbra, marcel).
c(delia, faramir). c(delia, george). c(delia, herodot).
c(erus, faramir). c(erus, george). c(erus, herodot).
```

*Soluție:*

```
frati(X, F) :- c(P, X), !, findall(Y, (c(P, Y), Y \= X), F). sau
frati(X, F) :- findall(Y, (c(P, X), c(P, Y), Y \= X), F1), sort(F1, F).
```

10. PROBLEMA (Poate fi implementată în orice limbaj studiat la PP.) Se urmărește implementarea unui *multi-map*, care este un tabel asociativ în care unei chei i se pot asocia oricără valori.

- (a) Descrieți reprezentarea *multi-map*-ului. Pentru Haskell, dați definiția tipului de date polimorfic. Definiți funcția/predicatul **lookup'**, care extrage lista tuturor valorilor asociate cu o cheie.  
(b) Definiți funcția/predicatul **insert'**, pentru adăugarea unei noi asocieri între o cheie și o valoare.  
(c) Definiți funcția/predicatul **map'**, care aplică o funcție/predicat pe fiecare valoare din *multi-map*.

NOTĂ: în Prolog, **map'** va aplica întotdeauna un același predicat **p(+VIn, -VOut)**.

*Soluție:*

Racket:

```
(define multimapExample '((a 1 2 3) (b 5 6 7) (c 7 8)))

(define (lookup k m) (cdr (assoc k m)))
(lookup 'b multimapExample)

(define (insert k v m)
  (let-values (((bef aft) (splitf-at m (λ(kv) (not (equal? (car kv) k))))))
    (if (null? aft)
        (cons (list k v) m)
        (append bef (list (cons k (cons v (cdar aft)))) (cdr aft))
      )))
(insert 'd 5 multimapExample)
(insert 'b 9 multimapExample)
```

```
(define (mmap f m)
  (map
    (lambda (kv) (cons (car kv) (map f (cdr kv))))
    m))
  (mmap add1 multimapExample))
```

Haskell:

```
data MultiMap k a = MM [(k, [a])] deriving (Eq, Show)
```

```
ins :: Eq k => k -> a -> MultiMap k a -> MultiMap k a
ins k a (MM lst) = MM $ case back of
```

```
[] -> (k, [a]) : front
(_, as) : back -> (k, a : as) : front ++ back
where
(front, back) = break ((== k) . fst) lst
```

```
map' :: (a -> b) -> MultiMap k a -> MultiMap k b
map' f (MM lst) = MM $ map (\(k, as) -> (k, map f as)) lst
```

```
test = ins 2 'z' $ ins 3 'c' $ ins 2 'b' $ ins 1 'a' $ MM []
```

Prolog:

```
lookup(K, MM, Values) :- member((K, Values), MM).
```

```
insert(K, V, MM, Out) :-
  select((K, L), MM, MM1), !,
  Out = [(K, [V|L]) | MM1].
insert(K, V, MM, [(K, [V]) | MM]).
```

```
f(V, V1) :- V1 is V + 1.
```

```
map(MM, Out) :-
  findall((K, L1),
  (member((K, L), MM),
   findall(E1, (member(E, L), f(E, E1)), L1)),
  Out).
```

# Examen PP varianta B — NOT EXAM MODE

31.05.2019

ATENȚIE: Aveți 2 ore · 1-9: 10p; 10: 30p · 100p pentru nota maximă · **Justificați** răspunsurile!

1. Reduceți expresia lambda  $E = (y \ (\lambda x. \lambda x. x \ (\lambda y. y \ y)))$

*Soluție:*

$$\rightarrow (y \ (\lambda x. \lambda x. x \ y)) \rightarrow (y \ \lambda x. x)$$

sau

$$\rightarrow (y \ \lambda x. x)$$

2. Se dă următorul cod Racket:

```
(define computation (λ () (equal? 5 5)))
(define (f x) (and (> x 5) (computation)))
(filter f '(1 3 5 7 9))
```

(a) De câte ori se apelează funcția `equal?` ?

(b) Rescrieți codul pentru `computation` și pentru `f` folosind promisiuni (pentru întârzierea lui `computation`) și răspundeți din nou la întrebarea (a).

*Soluție:*

(a) de 2 ori (pentru fiecare element mai mare decât 5)

```
(b) (define computation (delay (equal? 5 5)))
     (define (f x) (and (> x 5) (force computation)))
     (filter f '(1 3 5 7 9))
```

acum se apelează o singură dată, la prima evaluare a lui `computation`.

3. Datează fiind o listă de liste de numere LL, scrieți în Racket codul care produce sublista lui LL în care pentru toate elementele L suma elementelor este cel puțin egală cu produsul lor. E.g. pentru  $L = ((1 2 3) (1 2) (4 5) (.5 .5))$  rezultatul este  $((1 2 3) (1 2) (0.5 0.5))$ . Nu folosiți recursivitate explicită.

*Soluție:*

```
(filter (lambda (L) (>= (apply + L) (apply * L))) '((1 2 3) (1 2) (4 5) (.5 .5)))
```

4. Sintetizați tipul următoarei funcții în Haskell: `f = map (++)`

*Soluție:*

```
map :: (a -> b) -> [a] -> [b]
(++) :: [c] -> ([c] -> [c])
a = [c]
b = [c] -> [c]
f :: [[c]] -> [[c] -> [c]]
```

5. (a) Câte aplicații ale funcției de incrementare sunt calculate pentru evaluarea expresiei Racket `(length (map add1 '(1 2 3 4 5 6 7 8 9 10)))` ?

(b) Dar pentru expresia Haskell `length $ map (+ 1) [1 .. 10]` ?

*Soluție:*

(a) Toate elementele listei sunt evaluate, deci 10.

(b) Elementele listei nu sunt evaluate, deci 0.

6. Supraîncărcați în Haskell operatorii `(+)` și `(*)` pentru valori booleene, pentru a surprinde operațiile de *sau*, respectiv *și* logic.

*Soluție:*

```
instance Num Bool where
  (+) = (||)
  (*) = (&&)
```

7. Transformați propoziția „Nu mor caii când vor căinii.” în logică cu predicate de ordinul întâi, folosind predicatele `cainii_vor(când)` și `cainii_vor(când)`.

*Soluție:*

$\exists t. cainii\_vor(t) \wedge \neg caii\_mor(t)$  – există și momente când căinii vor dar caii nu mor sau  
 $\neg(\forall t. cainii\_vor(t) \Rightarrow caii\_mor(t))$  – nu este adevărat că oricând căinii vor automat caii mor

8. Se dă programul Prolog:

```
p(_, [], []).  
p(A, [A|B], B) :- !.  
p(A, [B|C], [B|D]) :- p(A, C, D).
```

Ce relație există între cele 3 valori  $X, Y, Z$ , dacă  $p(X, Y, Z)$  este adevărat?

*Soluție:*

Este predicatul `select`, iar dacă primul argument este nelegat face `select` la primul element. Predicatul `select(X, Y, Z)` este adevărat dacă  $X$  este un element din lista  $Y$ , iar  $Z$  este exact lista  $Y$ , în afară de elementul  $X$ .

9. Se dau următoarele relații genealogice prin predicatul `c(Parinte, Copil)`. Implementați predicatul `veri(X, V)`, care leagă  $V$  la lista de veri ai lui  $X$  (dacă există). De exemplu, pentru definițiile de mai jos, interogarea `veri(faramir, V)` leagă  $V$  la `[jenny, karl, ninel, octav]`.

```
c(alex, celia). c(alex, delia). c(alex, marcel).  
c(barbra, celia). c(barbra, delia). c(barbra, marcel).  
c(delia, faramir).  
c(ion, jenny). c(ion, karl). c(celia, jenny). c(celia, karl).  
c(marcel, ninel). c(marcel, octav).
```

*Soluție:*

```
veri(X, L) :- setof(V, P^B^U^(c(P, X), c(B, P), c(B, U), U\=P, c(U, V)), L). sau  
veri(X, L) :- findall(V, (c(P, X), c(B, P), c(B, U), U\=P, c(U, V)), L1), sort(L1, L).
```

10. PROBLEMA (Poate fi implementată în orice limbaj studiat la PP.) Se urmărește implementarea unui *hash set*, care reprezintă o mulțime grupând valorile în bucket-uri, fiecare bucket fiind unic determinat de hash-ul valorilor din bucket (toate valorile din bucket au același hash). Hashul unei valori va fi dat de funcția `hash`, respectiv predicatul `hash(+V, -Hash)`.

- (a) Descrieți reprezentarea *hash set*-ului. Pentru Haskell, dați definiția tipului de date polimorfic. Definiți funcția/predicatul `values'`, care extrage lista tuturor valorilor asociate cu un *hash*.
- (b) Definiți funcția/predicatul `insert'`, pentru adăugarea unei valori.
- (c) Definiți funcția/predicatul `map'`, care aplică o funcție/predicat pe fiecare valoare din *hash-set*.

NOTĂ: în Prolog, `map'` va aplica întotdeauna un același predicat `p(+VIn, -VOut)`.

*Soluție:*

Racket:

```
(define (hash int) (remainder int 2))  
(define hashExample '((1 3 5 7) (0 0 2 8)))  
  
(define (lookup hash m) (cdr (assoc hash m)))  
(lookup 1 hashExample)  
  
(define (insert v m)  
  (let ((k (hash v)))  
    (let-values (((bef aft) (splitf-at m (λ(kv) (not (equal? (car kv) k))))))  
      (if (null? aft)  
          (cons (list k v) m)  
          (append bef (list (cons k (cons v (cdar aft)))) (cdr aft))  
        ))))  
(insert 9 hashExample)  
(insert 9 (cdr hashExample))  
  
(define (mmmap f m)
```

```

(map
  (λ(kv) (cons (car kv) (map f (cdr kv))))
  m))
(mmap add1 hashExample)

Haskell:
class Hashable a where hash :: a -> Int -- nu a fost cerut în rezolvare
instance Hashable Int where hash x = mod x 2 -- nu a fost cerut în rezolvare

data HashSet a = HS [(Int, [a])] deriving (Eq, Show)

--ins :: Hashable a => a -> HashSet a -> HashSet a
--// tipul corect, dar mai greu de testat
ins :: Int -> HashSet Int -> HashSet Int
ins a (HS lst) = HS $ case back of
  [] -> (k, [a]) : front
  (_, as) : back -> (k, a : as) : front ++ back
  where
    k = hash a
    (front, back) = break ((== k) . fst) lst

map :: (Hashable a, Hashable b) => (a -> b) -> HashSet a -> HashSet b
map f (HS lst) = HS $ map (\(k, as) -> (k, map f as)) lst

test2 = ins 5 $ ins 2 $ ins 4 $ ins 8 $ HS []
test3 = map (+1) $ test2

```

Prolog:

```

hash(V, Hash) :- Hash is mod(V, 2).

lookup(Hash, HS, Values) :- member((Hash, Values), HS).

insert(V, HS, Out) :-
  hash(V, K),
  select((K, L), HS, HS1), !,
  Out = [(K, [V|L]) | HS1].
insert(V, HS, [(K, [V]) | HS]) :- hash(V, K).

% insert(5, [], H1), insert(7, H1, H2), insert(2, H2, H3).

f(V, V1) :- V1 is V + 1.

map(HS, Out) :-
  findall((K, L1),
    (member((K, L), HS),
      findall(E1, (member(E, L), f(E, E1)), L1)),
    Out).

```

Precizări:

- Fiecare subiect are câte 10p.
- Este suficientă rezolvarea a 10 subiecte din cele 12 pentru nota maximă.

1. Dați un exemplu de  $\lambda$ -expresie pentru care reducerea dreapta-stânga se termină, dar cea stânga-dreapta, nu. Justificați!

*Soluție.* .

Conform teoremei normalizării, nu există o astfel de expresie.  $\square$

2. Inversați o listă în Haskell, întrebuiuțând **funcțională** `foldl` și **fără** a utiliza recursivitate explicită. Solutiile care nu respectă cele două constrângeri **nu** vor fi punctate!

`foldl :: (a -> b -> a) -> a -> [b] -> a`

*Soluție.* .

`foldl (flip (:)) [] [1, 2, 3]`  $\square$

3. La ce se evaluează ultima expresie, în Scheme? Justificați!

```
1 (define f
2   (lambda (x)
3     (lambda (y)
4       (+ x y))))
5
6 (let* ([f-1 (f 1)]
7       [y 2]
8       [x ((f 1) (f-1 1))])
9 (f-1 3))
```

*Soluție.* .

La 4. În corpul lui `let*`, nicio altă variabilă în afară de `f-1` nu este folosită.  $\square$

4. Definiți obiectul `x` în Scheme, astfel încât expresia `(eq? x ○)` să se evalueze la #t dacă ○ este de forma `x, ((x)), (((x)))`, ... (număr **par** de aplicări), respectiv la #f dacă ○ este de forma `(x), ((x))`, ... (număr **impar** de aplicări).

*Soluție.* .

```
1 (define x (lambda () (lambda () x)))
```

$\square$

5. Căți parametri se vor calcula în procesul de evaluare a expresiei Scheme de mai jos? Justificați! (5p) Găsiți o reprezentare sugestivă pentru valoarea expresiei. (5p)

```
1 (let ([f (lambda (x y)
2           (lambda (z)
3             (+ x y z)))])
4   (f (+ 1 2) (+ 2 3)))
```

*Soluție.* .

Se vor evalua primii 2 parametri, datorită evaluării aplicative din Scheme. Valoarea este închiderea funcțională de mai jos:

$$\langle \lambda z. (+\ x\ y\ z); \{x \leftarrow 3, y \leftarrow 5\} \rangle$$

□

6. Sintetizați tipul funcției Haskell de mai jos. Justificați!

$$f\ x = (x\ f, x\ (f\ x))$$

*Soluție.* .

Este suficient să ne uităm la prima componentă a perechii.

```
1  f :: a -> b
2  x :: c -> d
3  a = c -> d
4  c = a -> b
5  a = (a -> b) -> d
```

Tip infinit!

□

7. Supraîncărcați în Haskell modalitatea de reprezentare sub formă de sir, a funcțiilor de cel puțin doi parametri, tipul primului fiind obținut prin utilizarea unui constructor de tip, unar.

*Soluție.* .

```
1 {-# LANGUAGE FlexibleInstances #-}
2 instance Show (t a -> b -> c) where
3     show _ = "cool_function"
```

Desigur că nu se cere și prima linie. Exemplu de funcție: (!!) :: [a] -> Int -> a.

□

8. Definiți în Haskell fluxul numerelor perfecte. Un număr este *perfect* dacă este egal cu suma divizorilor lui, diferenți de numărul însuși. Primele două numere perfecte sunt  $6 = 1 + 2 + 3$  și  $28 = 1 + 2 + 4 + 7 + 14$ .

*Soluție.* .

```
1 divisors x = [ d | d <- [ 1 .. x `div` 2 ], x `mod` d == 0 ]
2 perfectNumbers = [ x | x <- [ 2 .. ], sum (divisors x) == x ]
```

□

9. Transcrieți în logica cu predicate de ordin I următoarea propoziție:

*Există răspunsuri pe care dacă i le dau profesorului, acesta îmi dă câte un punct.*

*Soluție.* .

Utilizăm predicatul *da(expeditor, destinatar, obiect)*.

$$\exists r. (raspuns(r) \wedge (da(eu, profesor, r) \Rightarrow da(profesor, eu, 1)))$$

□

10. Scrieți un program Prolog, care interclasează două liste sortate crescător.

*Soluție.* .

```
1 merge([], L, L).
2 merge(L, [], L).
3 merge([H1 | T1], [H2 | T2], [H1 | L]) :- 
4     H1 =< H2, merge(T1, [H2 | T2], L), !.
5 merge([H1 | T1], [H2 | T2], [H2 | L]) :- 
6     merge([H1 | T1], T2, L).
```

□

11. Scrieți un algoritm Markov, care calculează funcția *succesor* pentru numere naturale. Numerele au reprezentare unară, sub formă unei secvențe de simboluri 1, de lungime egală cu valoarea numărului. De exemplu, numărul 3 este reprezentat prin secvența 111.

*Soluție.* .

În programul de mai jos, a și b sunt variabile de lucru.

```
1 Succesor()
2     a -> .
3     -> a1
4 end
```

Exemplu: 111 -3-> a1111 -2-> 1111.

□

12. Scrieți un program CLIPS, care elimină duplicatele consecutive dintr-o listă. De exemplu, pentru faptul (list a b b c c c a a b c c c c), se va genera rezultatul (list a b c a b c).

*Soluție.* .

```
1 (deffacts facts (list a b b c c c a a b c c c c))
2
3 (defrule remove
4     ?l <- (list $?pre ?x ?x $?post)
5     =>
6     (retract ?l)
7     (assert (list $?pre ?x $?post)))
```

□

## Examen PP – Seria 2CC

31.05.2014

NOTĂ: Fiecare subiect valorează 10 puncte. Este suficientă rezolvarea completă a 10 subiecte pentru nota maximă. Timpul de lucru este de 2 ore. Examenul este open-book. Pentru puctarea răspunsurilor este necesară **justificarea** acestora.

---

1. Reduceti expresia E la forma normală:  $E \equiv ((\lambda y.(\lambda x.\lambda y.x\ y)\ \lambda x.x)\ \Omega)$

*Soluție:*

$$((\lambda y.(\lambda x.\lambda y.x\ y)\ \lambda x.x)\ \Omega) \rightarrow ((\lambda x.\lambda y.x\ \lambda x.x)\ \Omega) \rightarrow (\lambda y.\lambda x.x\ \Omega) \rightarrow \lambda x.x$$

2. Implementați o funcție în Racket care ia o listă de numere L1 ca prim parametru și o listă de liste L2 ca al doilea parametru și întoarce acele liste din L2 ale căror lungimi se regăsesc în L1. Utilizați funcționale și nu utilizați recursivitate explicită – soluțiile care nu respectă cele două constrângeri nu vor fi punctate.

Exemplu:  $(f\ '(1\ 2\ 3\ 4)\ '((4\ 5\ 6)\ ()\ (a\ b))) \rightarrow '((4\ 5\ 6)\ (a\ b))$

*Soluție:*

$$(\lambda (L1\ L2)\ (filter\ (\lambda (l)\ (member\ (length\ l)\ L1))\ L2))$$

3. La ce se evaluatează următoarea expresie în Racket? (cu  $() \equiv []$ )

$(let* [(x\ 1)\ (y\ 2)\ (f\ (delay\ (\lambda (y)\ (+\ x\ y))))]\ (let\ [(x\ 5)]\ ((force\ f)\ x)))$

*Soluție:*

$$6. x=5 + x=1.$$

4. Scrieți în Haskell o funcție care realizează produsul cartezian a două mulțimi oarecare (liste) A și B. Utilizați facilitățile oferite de limbaj. Care este tipul funcției create?

*Soluție:*

$$\text{cart } a\ b = [(x,\ y) \mid x \leftarrow a,\ y \leftarrow b]$$

$$\text{cart} :: [t] \rightarrow [t] \rightarrow [(t,\ t)]$$

5. Construiți în Haskell fluxul puterilor lui 2.

*Soluție:*

$$\text{fluxus} = 1 : \text{map} (*\ 2) \text{ fluxus}$$

6. Sintetizați tipul funcției f în Haskell:

$$f\ x = [(y,\ y) \mid (y,\ z,\ t) \leftarrow x,\ y == z]$$

*Soluție:*

$$f :: Eq\ t1 \Rightarrow [(t1,\ t1,\ t)] \rightarrow [(t1,\ t1)]$$

7. Supraîncărcați în Haskell afișarea funcțiilor care au ca parametru un număr, afișându-se valoarea funcției în punctul 0.

*Soluție:*

$$\text{instance } (\text{Show}\ b,\ \text{Num}\ a) \Rightarrow \text{Show}\ (a \rightarrow b) \text{ where show}\ f = \text{show}\ (f\ 0)$$

8. Traduceți în logica cu predicate de ordinul I următoarea propoziție:

*Cine tace e mai înțelept decât cine vorbește.*

*Soluție:*

$$\forall x.\forall y.(tace(x) \wedge vorbeste(y)) \Rightarrow mai\_intelept(x,y)$$

9. Folosiți rezoluția pentru a demonstra că, din moment ce toți oamenii sunt muritori, Socrate, om el însuși, este muritor. Folosiți predicatele *om* și *muritor*.

*Soluție:*

Premise:  $\forall x. om(x) \Rightarrow muritor(x)$  ;  $om(Socrate)$

Concluzia:  $muritor(Socrate)$  (negată:  $\neg muritor(Socrate)$ )

în FNC:  $\{\neg om(x), muritor(x)\}, \{om(Socrate)\}, \{\neg muritor(Socrate)\}$

Pas de rezoluție cu rezolvent  $om(x)\{Socrate/x\} \rightarrow \{muritor(Socrate)\}$

Pas de rezoluție cu rezolvent  $muritor(Socrate) \rightarrow \{\}$

10. Ce rezultat are în Prolog evaluarea lui  $p(L, X)$ , cu L o listă și X nelegat:

$r([], _).$

$r([H|T], X) :- member(H, X), r(T, X).$

$p(L, X) :- length(L, N), length(X, N), r(L, X).$

*Soluție:*

Lista X are aceeași lungime ca și L și aceeași membri, în orice ordine (diversele soluții pentru X sunt permutările listei L).

11. Scrieți un predicat Prolog **up** (și eventual predicate ajutătoare) care identifică secvențele (cel puțin două elemente) strict crescătoare dintr-o listă. Exemplu:

$up([5, 1, 2, 3, 2, 3, 1, 1, 0, 9, 10], LS) \rightarrow LS = [1, 2, 3, 2, 3, 0, 9, 10]$

*Soluție:*

$up([], []).$

$up([H, H1 | T], [H, H1 | LS]) :- H1 > H, !, up(T, H1, LS).$

$up([_ | T], LS) :- up(T, LS).$

$up([], _, []) :- !.$

$up([H | T], E, [H | LS]) :- H > E, !, up(T, H, LS).$

$up(L, _, LS) :- up(L, LS).$

12. O transmisiune de date transmite câte doi biți de date urmați de un bit de control (suma modulo 2 a celor doi biți transmiși anterior). Scrieți un algoritm Markov care identifică secvențele eronate (bit de control greșit) și le marchează cu trei de x.

Exemplu: 101010110000111 → 101xxx110000xxx

*Soluție:*

a000 → 000a

a011 → 011a

a101 → 101a

a110 → 110a

ag<sub>1</sub>g<sub>2</sub>g<sub>3</sub> → xxxa

a → .

→ a

A

## Examen PP – Seria 2CC

31.05.2014

NOTĂ: Fiecare subiect valorează 10 puncte. Este suficientă rezolvarea completă a 10 subiecte pentru nota maximă. Timpul de lucru este de 2 ore. Examenul este open-book. Pentru puctarea răspunsurilor este necesară **justificarea** acestora.

---

1. Reduceti expresia E la forma normală:  $E \equiv ((\lambda y.(\lambda x.\lambda y.x\ y)\ \lambda x.x)\ \Omega)$

*Soluție:*

$$((\lambda y.(\lambda x.\lambda y.x\ y)\ \lambda x.x)\ \Omega) \rightarrow ((\lambda x.\lambda y.x\ \lambda x.x)\ \Omega) \rightarrow (\lambda y.\lambda x.x\ \Omega) \rightarrow \lambda x.x$$

2. La ce se evaluatează următoarea expresie în Racket? (cu  $() \equiv []$ )

```
(let* [(x 3) (y 4) (f (delay (λ (y) (+ x y))))] (let [(x 1)] ((force f) x)))
```

*Soluție:*

$$4. x=1 + x=3.$$

3. Implementați o funcție în Racket care ia o listă de numere L1 ca prim parametru și o listă de liste L2 ca al doilea parametru și întoarce acele liste din L2 ale căror lungimi nu se regăsesc în L1. Utilizați funcționale și nu utilizați recursivitate explicită – soluțiile care nu respectă cele două constrângeri nu vor fi punctate.

Exemplu:  $(f '(1 0 4 5) '((4 5 6) () (a b))) \rightarrow '((4 5 6) (a b))$

*Soluție:*

```
(λ (L1 L2) (filter (λ (l) (not (member (length l) L1))) L2))
```

4. Sintetizați tipul funcției f în Haskell:

```
f x = [(z, z) | (y, z, t) <- x, y == z]
```

*Soluție:*

```
f :: Eq t1 => [(t1, t1, t)] -> [(t1, t1)]
```

5. Suprăîncărcați în Haskell afișarea funcțiilor care au ca parametru un număr, afișându-se valoarea funcției în punctul 1.

*Soluție:*

```
instance (Show b, Num a) => Show (a -> b) where show f = show (f 1)
```

6. Construiți în Haskell fluxul puterilor lui 4.

*Soluție:*

```
fluxus = 1 : map (* 4) fluxus
```

7. Scrieți în Haskell o funcție care realizează produsul cartezian a două mulțimi oarecare (liste) A și B. Utilizați facilitățile oferite de limbaj. Care este tipul funcției create?

*Soluție:*

```
cart a b = [(x, y) | x <- a, y <- b]
```

```
cart :: [t] -> [t1] -> [(t, t1)]
```

8. Traduceți în logica cu predicate de ordinul I următoarea propoziție:

*Cine tace e mai înțelept decât cine vorbește.*

*Soluție:*

$$\forall x.\forall y.(tace(x) \wedge vorbeste(y)) \Rightarrow mai\_intelept(x, y)$$

9. Folosiți rezoluția pentru a demonstra că, din moment ce toți oamenii sunt muritori, Socrate, om el însuși, este muritor. Folosiți predicatele *om* și *muritor*.

*Soluție:*

Premise:  $\forall x. om(x) \Rightarrow muritor(x)$  ;  $om(Socrate)$

Concluzia:  $muritor(Socrate)$  (negată:  $\neg muritor(Socrate)$ )

în FNC:  $\{\neg om(x), muritor(x)\}, \{om(Socrate)\}, \{\neg muritor(Socrate)\}$

Pas de rezoluție cu rezolvent  $om(x)\{Socrate/x\} \rightarrow \{muritor(Socrate)\}$

Pas de rezoluție cu rezolvent  $muritor(Socrate) \rightarrow \{\}$

10. Scrieți un predicat Prolog **down** (și eventual predicate ajutătoare) care identifică secvențele (cel puțin două elemente) strict descrescătoare dintr-o listă. Exemplu:  
 $down([5, 1, 2, 3, 2, 1, 1, 10, 9, 10], LS) \rightarrow LS = [5, 1, 3, 2, 1, 10, 9]$

*Soluție:*

$down([], [])$ .

$down([H, H1 | T], [H, H1 | LS]) :- H1 < H, !, down(T, H1, LS)$ .

$down([_ | T], LS) :- down(T, LS)$ .

$down([], _, []) :- !$ .

$down([H | T], E, [H | LS]) :- H < E, !, down(T, H, LS)$ .

$down(L, _, LS) :- down(L, LS)$ .

11. Ce rezultat are în Prolog evaluarea lui **transformer(L, X)**, cu L o listă și X nelegat:  
 $processor([], _)$ .  
 $processor([H|T], X) :- member(H, X), processor(T, X)$ .  
 $processor(L, X) :- length(L, N), length(X, N), processor(L, X)$ .

*Soluție:*

Lista X are aceeași lungime ca și L și aceeași membri, în orice ordine (diversele soluții pentru X sunt permutările listei L).

12. O transmisiune de date transmite câte doi biți de date urmați de un bit de control (suma modulo 2 a celor doi biți transmiși anterior). Scrieți un algoritm Markov care identifică secvențele eronate (bit de control greșit) și le marchează cu trei de x.  
Exemplu: 101010110000111 → 101xxx110000xxx

*Soluție:*

a000 → 000a

a011 → 011a

a101 → 101a

a110 → 110a

ag<sub>1</sub>g<sub>2</sub>g<sub>3</sub> → xxxa

a → .

→ a

# Examen PP – Seria 2CC

11.06.2015

ATENȚIE: Aveți 2 ore . 10p per subiect . 100p necesare pentru nota maximă . **Justificați răspunsurile!**

---

1. Reduceti la forma normală expresia:

$$(\lambda y.((\lambda x.\lambda y.x\ y)\ \Omega)\ \lambda x.\lambda y.y)$$

*Soluție:*

$$\begin{aligned} & (\lambda y.((\lambda x.\lambda y.x\ y)\ \Omega)\ \lambda x.\lambda y.y) \\ \rightarrow & ((\underline{\lambda x}.\lambda y.x\ \underline{\lambda x.\lambda y.y})\ \Omega) \\ \rightarrow & (\lambda y.\lambda x.\lambda y.y\ \underline{\Omega}) \\ \rightarrow & \lambda x.\lambda y.y \end{aligned}$$

2. Scrieți o funcție setN în Racket care primește două liste L1 și L2 (fără duplicate) ca argumente și întoarce o listă care este intersecția celor două liste, luate ca multimi (rezultatul nu trebuie să conțină duplicate).

*Soluție:*

```
(define (setU L1 L2)
  (cond
    ((null? L1) L2)
    ((member (car L1) L2) (setU (cdr L1) L2))
    (else (cons (car L1) (setU (cdr L1) L2))))
  )) sau
(define (setU2 L1 L2) (foldr (λ (x L)
  (if (member x L) L (cons x L))) '() (append L1 L2)))
```

3. Date fiind funcțiile E și F și următorul cod care considerăm că se execută fără erori, de câte ori sunt evaluate fiecare dintre cele două funcții, și la ce linii din cod se fac evaluările?

1. (define fmic (λ (a)
2. (let [ (f (F a)) (g (delay (E a))) ]
3. f ) ))
4. (fmic (E 'argument))

*Soluție:*

E la 4, F la 2

4. Sintetizați tipul funcției Haskell următoare:  $f\ x = f\ (f\ x)$

*Soluție:*

```
f :: a -> b
x :: a
(f (f x)) :: b
-----
f :: c -> d
(f x) :: c
d = b
-----
f :: e -> g
x :: e = a
g = c
din f: a = c = e și b = d = g
f :: a -> a
-----
f :: t -> t
```

5. Instanțiați în Haskell clasa `Ord` pentru liste. Ordonarea listelor va fi dată de compararea primului element din fiecare listă. E.g. `[1, 2, 3] < [2, 3, 4]` pentru că `1 < 2`.

*Soluție:*

NOTĂ: Pentru ca implementarea să compileze am folosit aici `MyOrd` și `#<=` în loc de `Ord` și `<=`, definite astfel: `class Eq a => MyOrd a where (#<=) :: a -> a -> Bool`. Solutia cerută, (dar cu `Ord` și `<=` în loc de `MyOrd` și `#<=`), era:

```
instance Ord a => MyOrd [a] where
    (h1:_ ) #<= (h2:_ ) = h1 <= h2
```

6. Scrieți o funcție Haskell care elimină dintr-o listă elementele care apar de mai multe ori. E.g. `[1, 2, 3, 2, 3, 5] -> [1, 5]`.

*Soluție:*

```
nodups [] = []
nodups l = head l : nodups (filter (not . (== head l)) $ tail l)
```

7. Care este fluxul `s` pentru care este adevărat:

```
(take 10 $ zipWith (+) s [1, 1..]) == (take 10 $ tail s)
```

*Soluție:*

Naturals

8. Traduceți în logica cu predicate de ordinul I următoarea propoziție:

*Cine se scuză, se acuză.*

*Soluție:*

$$\forall x. scuza(x, x) \Rightarrow acuza(x, x)$$

9. Stind că în mijlocul Dunării există insula AdaKaleh, și există o trecere de pe malul drept pe insulă, și una de pe insulă pe malul stâng, iar trecerea este tranzitivă, folosiți rezoluția pentru a demonstra că se poate trece de pe malul drept pe malul stâng (folosiți predicatul `trece(loc, altLoc)`).

*Soluție:*

$$trece(MStang, AdaKaleh) \text{ (1)}$$

$$trece(AdaKaleh, MDrept) \text{ (2)}$$

$$\forall x \forall y \forall z. trece(x, y) \wedge trece(y, z) \Rightarrow trece(x, z) \text{ (tranzitivitate)}$$

$$\rightarrow \{\neg trece(x, y) \vee \neg trece(y, z) \vee trece(x, z)\} \text{ (3)}$$

$$\neg trece(MStang, MDrept) \text{ (4) (concluzie negată)}$$

(1) rezolvă cu (3), substituție  $\{MStang/x, AdaKaleh/y\}$

$$\rightarrow \neg trece(AdaKaleh, z) \vee trece(MStang, z) \text{ (5)}$$

(2) rezolvă cu (5), substituție  $\{MDrept/z\} \rightarrow trece(MStang, MDrept)$  (6)

(6) rezolvă cu (4)  $\rightarrow$  clauza vidă.

10. Implementați în Prolog un predicat având semnătura `mapF(+L, -LO)`, care să fie echivalent cu expresia Haskell `(map f)`, știind că există deja definit un predicat `f(+X, -X0)`.

*Soluție:*

$$\text{mapF(L, LO):- findall(XO, (member(X, L), f(X, XO)), LO).}$$

11. Câte soluții are interogarea `p(L, [1, 2, 3])` în condițiile în care avem definiția de mai jos? Ce formă au aceste soluții?

$$p(D, [A, B, C]) :- \text{member}(A, D), \text{member}(B, D), \text{member}(C, D).$$

*Soluție:*

O infinitate. Soluțiile sunt liste care conțin 1, 2 și 3 (în ordine) și un număr  $\geq 0$  de elemente neinstantiate.

12. Scrieți un algoritm Markov care lucrează pe un sir de simboluri din multimea A și înlocuiește fiecare grup de două simboluri identice cu o singură apariție a simbolului. Grupurile identificate trebuie să fie disjuncte. De exemplu, pentru sirul 0100110110001 se obține 010101001.

*Soluție:*

1. Dedup(); Ab g<sub>1</sub>
2. ag<sub>1</sub>g<sub>1</sub> → g<sub>1</sub>a
3. ag<sub>1</sub> → g<sub>1</sub>a
4. a → .
5. → a

^

# Examen PP – Seria 2CC

11.06.2015

ATENȚIE: Aveți 2 ore . 10p per subiect . 100p necesare pentru nota maximă . **Justificați răspunsurile!**

1. Reduceti la forma normală expresia:

$$(((\lambda x.\lambda y.\lambda z.y \ \lambda x.x) \ (\lambda z.\lambda t.z \ z)) \ \Omega)$$

*Soluție:*

$$\begin{aligned} & (((\lambda x.\lambda y.\lambda z.y \ \lambda x.x) \ (\lambda z.\lambda t.z \ z)) \ \Omega) \\ \rightarrow & (((\underline{\lambda x}.\lambda y.\lambda z.y \ \underline{\lambda x}.x) \ (\lambda z.\lambda t.z \ z)) \ \Omega) \\ \rightarrow & ((\underline{\lambda y}.\lambda z.y \ (\lambda z.\lambda t.z \ z)) \ \Omega) \\ \rightarrow & (\underline{\lambda w}.\lambda z.\lambda t.z \ z) \ \Omega \\ \rightarrow & (\underline{\lambda v}.\lambda t.v \ z) \\ \rightarrow & \lambda t.z \end{aligned}$$

2. Scrieți o funcție setU în Racket care primește două liste L1 și L2 (fără duplicate) ca argumente și întoarce o listă care este reuniunea celor două liste, luate ca multimi (rezultatul nu trebuie să conțină duplicate).

*Soluție:*

```
(define (setN L1 L2)
  (cond
    ((null? L1) '())
    ((member (car L1) L2) (cons (car L1) (setN (cdr L1) L2)))
    (else (setN (cdr L1) L2)))
  )) sau
(define (setN2 L1 L2) (filter (λ (x) (member x L2)) L1))
```

3. Date fiind funcțiile E și F și următorul cod care considerăm că se execută fără erori, de câte ori sunt evaluate fiecare dintre cele două funcții, și la ce linii din cod se fac evaluările?

1. (define gmic (λ (a)
2. (let [ (f (delay (F a))) (x (g a)) ]
3. f ) ))
4. (gmic (E 'argument))

*Soluție:*

E la 4, F niciodată.

4. Sintetizați tipul funcției Haskell următoare:  $f \ x = x \ (f \ x)$

*Soluție:*

$f :: a \rightarrow b$

$x :: a$

$(x \ (f \ x)) :: b$

-----

$x :: c \rightarrow d$

$(f \ x) :: c$

$d = b$

-----

$f :: e \rightarrow g$

$e = a = c \rightarrow d$

$g = b = c = d$

$f :: a \rightarrow b = (b \rightarrow b) \rightarrow b$

-----

$f :: (t \rightarrow t) \rightarrow t$

5. Instanțiați în Haskell clasa `Ord` pentru perechi. Ordinea perechilor va fi dată de compararea celui de-al doilea element din pereche. E.g.  $(1, 2) > (2, 0)$  (pentru că  $2 > 0$ ).

*Soluție:*

NOTĂ: Pentru ca implementarea să compileze am folosit aici `MyOrd` și `#<=` în loc de `Ord` și `<=`, definite astfel: `class Eq a => MyOrd a where (#<=) :: a -> a -> Bool.` Solutia cerută, (dar cu `Ord` și `<=` în loc de `MyOrd` și `#<=`), era:

```
instance (Eq a, Ord b) => MyOrd (a, b) where
    ( _, y1) #<= ( _, y2) = y1 <= y2
```

6. Scrieți o funcție Haskell care păstrează dintr-o listă doar valorile care apar de mai multe ori. E.g.  $[1, 2, 3, 2, 3] \rightarrow [2, 3]$ .

*Soluție:*

```
dups [] = []
dups (h:t)
| elem h t = h : dups (filter (/= h) t)
| otherwise = dups t
```

7. Care este fluxul `s` pentru care este adevărat:

```
(take 10 $ zipWith (+) s (tail s)) == (take 10 $ (tail . tail) s)
```

*Soluție:*

Fibonacci

8. Traduceți în logica cu predicate de ordinul I următoarea propoziție:

*Cine are carte, are parte.*

*Soluție:*

$$\forall x. \text{are}(x, \text{Carte}) \Rightarrow \text{are}(x, \text{Parte})$$

9. Știind că elefantul este mai mare decât leul, și leul este mai mare decât șoricelul, iar relația de 'mai mare' este tranzitivă, folosiți rezoluția pentru a demonstra că elefantul este mai mare decât șoricelul

*Soluție:*

*maiMare(Elefant, Leu) (1)*

*maiMare(Leu, Soricel) (2)*

$\forall x \forall y \forall z. \text{maiMare}(x, y) \wedge \text{maiMare}(y, z) \Rightarrow \text{maiMare}(x, z)$  (tranzitivitate)

$\rightarrow \{\neg \text{maiMare}(x, y) \vee \neg \text{maiMare}(y, z) \vee \text{maiMare}(x, z)\}$  (3)

$\neg \text{maiMare}(\text{Elefant}, \text{Soricel})$  (4) (concluzie negată)

(1) rezolvă cu (3), substituție {*Elefant*/x, *Leu*/y}

$\rightarrow \neg \text{maiMare}(\text{Leu}, z) \vee \text{maiMare}(\text{Elefant}, z)$  (5)

(2) rezolvă cu (5), substituție {*Soricel*/z}  $\rightarrow \text{maiMare}(\text{Elefant}, z)$  (6)

(6) rezolvă cu (4)  $\rightarrow$  clauza vidă.

10. Implementați în Prolog un predicat având semnătura `filterF(+L, -LO)`, care să fie echivalent cu expresia Haskell (`filter f`), știind că există deja definit un predicat `f(+X)`.

*Soluție:*

`filterF(L, LO):- findall(X, (member(X, L), f(X)), LO).`

11. Câte soluții are interogarea `p([1, 2, 3], L)` în condițiile în care avem definiția de mai jos? Ce formă au aceste soluții?

`p(D, [A, B, C]) :- member(A, D), member(B, D), member(C, D).`

*Soluție:*

27. Sunt toate combinațiile de 1,2,3, inclusiv în care un element apare de mai multe ori.

12. Scrieți un algoritm Markov care lucrează pe un sir de simboluri din mulțimea A și înlocuiește fiecare grup de două sau mai multe simboluri identice consecutive cu o singură apariție a simbolului. De exemplu, pentru sirul 0100110110001 se obține 01010101.

*Soluție:*

1. Dedup(); Ab g<sub>1</sub>
2. g<sub>1</sub>g<sub>1</sub>-> g<sub>1</sub>
3. -> .

<

# Examen PP – Seria 2CC

11.06.2016

ATENȚIE: Aveți 2 ore . 10p per subiect . 100p necesare pentru nota maximă . **Justificați răspunsurile!**

- Ilustrați cele două posibile secvențe de reducere pentru expresia:  $(\lambda y.(\lambda x.\lambda y.x\ y)\ 2)$

*Soluție:*

- $(\underline{\lambda y}.(\lambda x.\lambda y.x\ \underline{y})\ 2) \xrightarrow[\beta]{stanga-dreapta} (\underline{\lambda x}.\lambda y.x\ 2) \rightarrow_\beta \lambda y.2$
- $(\lambda y.(\lambda x.\underline{\lambda y}.x\ y)\ 2) \rightarrow_\alpha (\lambda y.(\underline{\lambda x}.\lambda z.x\ y)\ 2) \xrightarrow[\beta]{dreapta-stanga} (\lambda y.\lambda z.y\ 2) \rightarrow_\beta \lambda z.2$

- Implementați în Racket o funcție `myAndMap` care să aibă un comportament similar cu `andmap` – primește o listă și întoarce o valoare booleană egală cu rezultatul operației `and` pe elementele listei. Folosiți cel puțin o funcțională. Nu folosiți `andmap`.

*Soluție:*

```
(define (myAndMap L) (foldl (λ (x y) (and x y)) #t L)) (am acceptat și foldl/r direct cu and, soluție cu filter, etc)
```

- Ce întoarce următoarea expresie în Racket? Justificați!

```
(let ((n 2))
  (letrec ((f (lambda (n)
    (if (zero? n) 1 (* n (f (- n 1)))))))
    (f 5)))
)
```

*Soluție:*

Este factorial.  $5! = 120$ . `n` din let nu are niciun efect pentru că în cod se folosește `n` legat de `lambda`.

- Cum se poate îmbunătăți următorul cod Racket pentru ca funcția `calcul-complex` să se evalueze doar atunci când este necesar, adică doar atunci când `variant` este fals (fără a o muta apelul lui `calcul-complex` în interiorul lui `calcul`) ?

1. `(define (calcul x y z) (if x y z))`
2. `(define (test variant) (calcul variant 2 (calcul-complex 3)))`

*Soluție:*

1. `(define (calcul x y z) (if x y (force z)))`
2. `(define (test variant) (calcul variant 2 (delay (calcul-complex 3))))`

Se poate și folosind închidere lambda și (`z`), `if` peste apelul lui `calcul-complex`, sau chiar `quote` și `eval`.

- Sintetizați tipul funcției `f` în Haskell:  $f\ x\ y = (y\ x)\ x$

*Soluție:*

```
f :: a → b → c
y :: d → e
b = d → e (y este argumentul lui f)
d = a (y ia ca argument pe x)
e = a → c (valoarea întoarsă de y)
⇒ f :: a → (a → a → c) → c
```

- Instanțiați în Haskell clasa `Eq` pentru tripluri, considerând că  $(a_1, a_2, a_3)$  este egal cu  $(b_1, b_2, b_3)$  dacă  $a_1 == b_1$  și  $a_2 == b_2$ .

*Soluție:*

```
instance (Eq a, Eq b) => Eq (a, b, c) where (a1, a2, _) == (b1, b2, _) = (a1 == b1) && (a2 == b2)
```

7. Implementați în Haskell, fără a utiliza recursivitate explicită, funcția `setD` care realizează diferența a două mulțimi `a` și `b` ( $a \setminus b$ ) date ca liste (fără duplicate). Care este tipul funcției?

*Soluție:*

```
setD a b = [x | x <- a, not $ elem x b]
sau
setD a b = filter (not . (flip elem) b) a
setD :: Eq t => [t] -> [t] -> [t]
```

8. Traduceți în logica cu predicate de ordinul întâi propoziția: *Orice naș își are nașul.*

*Soluție:*

$$\forall x \forall y. nas(x, y) \Rightarrow \exists z. nas(z, x)$$

sau

$$\forall x. (\exists y. nas(x, y)) \Rightarrow \exists z. nas(z, x)$$

9. Știind că  $\forall x. Trezit(x, Dimineata) \Rightarrow \forall y. AjungeLa(x, y)$  și că  $Trezit(Eu, Dimineata)$ , demonstrați, folosind **metoda rezoluției**, că  $AjungeLa(Eu, Examen)$ .

*Soluție:*

FNC:

$$\neg Trezit(x, Dimineata) \vee Ajunge(x, y) \quad (1)$$

$$Trezit(Eu, Dimineata) \quad (2)$$

$$\neg AjungeLa(Eu, Examen) \quad (3) \text{ (negarea concluziei)}$$

Rezoluție:

(1) rezolvă cu (2), cu rezolventul  $Trezit(Eu, Dimineata)$ , sub substituția  $x \leftarrow Eu$  obținem clauza  $AjungeLa(Eu, y)$  (4)

(3) rezolvă cu (4), sub substituția  $y \leftarrow Examen$ , rezultă clauza vidă.

10. Care este efectul aplicării predicatului `p` asupra listelor `L1` și `L2` (la ce este legat argumentul `R` în apelul `p(L1, L2, R)`?)

$$p(A, [], A) . p(A, [E|T], [E|R]) :- p(A, T, R).$$

*Soluție:*

$$R = L2 ++ L1$$

11. Implementați un algoritm Markov care primește un sir de simboluri 0 și 1 și verifică dacă sirul începe cu 0 și se termină cu 1 și, în caz afirmativ, adaugă la sfârșitul sirului simbolurile “ok”, altfel nu schimbă sirul cu nimic. Exemplu: 010111011 → 010111011ok ; 010 → 010 ; 1010 → 1010

*Soluție:*

1. Check(); {0, 1} g<sub>1</sub>
2. a0→0b
3. bg<sub>1</sub>→g<sub>1</sub>b
4. 1b→1okb
5. b→.
6. a→.
7. →a

12. Explicați care dintre următoarele apeluri dă eroare și care nu, și justificați pentru fiecare:

1. (if #t 5 (/ 2 0)) (Racket)
2. (let ((f (λ (x y) x))) (f 5 (/ 2 0))) (Racket)
3. let f x y = x in f 5 (div 2 0) (Haskell)
4. X = 2 / 0, Y = X. (Prolog)

*Soluție:*

1. Nu este eroare → if este funcție nestriictă.

2. Eroare, din cauza evaluării aplicative.
3. Nu este eroare, datorită evaluării lenșe (y nu este folosit).
4. Nu este eroare, = nu evluează calculele aritmetice.

# Examen PP – Seria 2CC

11.06.2016

ATENȚIE: Aveți 2 ore . 10p per subiect . 100p necesare pentru nota maximă . **Justificați răspunsurile!**

- Ilustrați cele două posibile secvențe de reducere pentru expresia:  $(\lambda x.(\lambda y.\lambda x.y\ x)\ 5)$

*Soluție:*

- $(\underline{\lambda x}.(\lambda y.\lambda x.y\ \underline{x})\ 5) \xrightarrow{\text{stanga-dreapta}}_{\beta} (\underline{\lambda y}.\lambda x.\underline{y}\ 5) \rightarrow_{\beta} \lambda x.5$
- $(\lambda x.(\lambda y.\underline{\lambda x.y}\ x)\ 5) \rightarrow_{\alpha} (\lambda x.(\lambda y.\underline{\lambda z.y}\ x)\ 5) \xrightarrow{\text{dreapta-stanga}}_{\beta} (\lambda x.\lambda z.x\ 5) \rightarrow_{\beta} \lambda x.5$

- Implementați în Racket o funcție `myOrMap` care să aibă un comportament similar cu `ormap` – primește o listă și întoarce o valoare booleană egală cu rezultatul operației `or` pe elementele listei. Folosiți cel puțin o funcțională. Nu folosiți `ormap`.

*Soluție:*

```
(define (myOrMap L) (foldl (λ (x y) (or x y)) #f L)) (am acceptat și foldl/r direct cu or, soluție cu filter, etc)
```

- Ce întoarce următoarea expresie în Racket? Justificați!

```
(letrec ((f (lambda (n)
  (let ((n (- n 1)))
    (if (eq? n -1) 1 (* (+ n 1) (f n)))))))
  (f 5))
)
```

*Soluție:*

Este factorial.  $5! = 120$ .

- Cum se poate îmbunătăți următorul cod Racket pentru ca funcția `calcul-complex` să se evalueze doar atunci când este necesar, adică doar atunci când `variant` este fals (fără a o muta apelul lui `calcul-complex` în interiorul lui `calcul`) ?

1. `(define (calcul x y z) (if x y z))`
2. `(define (test variant) (calcul variant 2 (calcul-complex 3)))`

*Soluție:*

1. `(define (calcul x y z) (if x y (force z)))`
2. `(define (test variant) (calcul variant 2 (delay (calcul-complex 3))))`

Se poate și folosind închidere lambda și `(z)`, `if` peste apelul lui `calcul-complex`, sau chiar `quote` și `eval`.

- Sintetizați tipul funcției `f` în Haskell:  $f\ x\ y = x\ (y\ x)$

*Soluție:*

```
f :: a → b → c
x :: d → e
y :: g → h
d = h (x ia valoarea întoarsă de y)
e = c (f întoarce valoarea întoarsă de x)
a = g = d → e (y și f iau ca argument pe x)
b = g → h (tipul lui x în f)
⇒ b = (d → c) → d; a = d → c
⇒ f :: (d → c) → ((d → c) → d) → c
```

- Instanțiați în Haskell clasa `Ord` pentru tripluri (știind că `Eq` este deja instanțiată), considerând că  $(a_1, a_2, a_3)$  este mai mic decât  $(b_1, b_2, b_3)$  dacă  $a_1 < b_1$ .

*Soluție:*

```
instance Ord a => Ord (a, b, c) where (a1, _, _) < (b1, _, _) = a1 < b1
```

7. Implementați în Haskell, fără a utiliza recursivitate explicită, funcția `setN` care realizează intersecția a două multimi a și b date ca liste (fără duplicate). Care este tipul funcției?

*Soluție:*

```
setN a b = [x | x <- a, elem x b]
sau
setN a b = filter ((flip elem) b) a
setN :: Eq t => [t] -> [t] -> [t]
```

8. Traduceți în logica cu predicate de ordinul întâi propoziția: *Orice copil are o mamă.*

*Soluție:*

 $\forall x.\text{copil}(x) \Rightarrow \exists y.\text{mama}(y, x)$ 

9. Știind că  $\forall x.\text{Are}(x, \text{Carte}) \Rightarrow \forall y.\text{Are}(x, y)$  și că  $\text{Are}(\text{Eu}, \text{Carte})$ , demonstrați, folosind **metoda rezoluției**, că  $\text{Are}(\text{Eu}, \text{Parte})$ .

*Soluție:*

FNC:

$$\begin{aligned} &\neg\text{Are}(x, \text{Carte}) \vee \text{Are}(x, y) \quad (1) \\ &\text{Are}(\text{Eu}, \text{Carte}) \quad (2) \\ &\neg\text{Are}(\text{Eu}, \text{Parte}) \quad (3) \quad (\text{negarea concluziei}) \end{aligned}$$

Rezoluție:

$$\begin{aligned} &(1) \text{ rezolvă cu (2), cu rezolventul } \text{Are}(\text{Eu}, \text{Carte}), \text{ sub substituția } x \leftarrow \text{Eu} \\ &\text{obținem clauza } \text{Are}(\text{Eu}, y) \quad (4) \\ &(3) \text{ rezolvă cu (4), sub substituția } y \leftarrow \text{Carte}, \text{ rezultă clauza vidă.} \end{aligned}$$

10. Care este efectul aplicării predicatului `p` asupra listelor `L1` și `L2` (la ce este legat argumentul `R` în apelul `p(L1, L2, R)`?)

 $p([], A, A). \quad p([E|T], A, [E|R]) :- p(T, A, R).$ 

*Soluție:*

 $R = L1 ++ L2$ 

11. Implementați un algoritm Markov care primește un sir de simboluri 0 și 1 și verifică dacă sirul începe cu 1 și se termină cu 0 și, în caz afirmativ, adaugă la sfârșitul sirului simbolurile "ok", altfel nu schimbă sirul cu nimic. Exemple: 1110100 → 1110100ok ; 0101 → 0101 ; 010 → 010 ; 1010 → 1010ok

*Soluție:*

1. Check(); {0, 1} g<sub>1</sub>
2. a1→1b
3. bg<sub>1</sub>→g<sub>1</sub>b
4. 0b→0okb
5. b→.
6. a→.
7. →a

12. Explicați care dintre următoarele apeluri dă eroare și care nu, și justificați pentru fiecare:

1. (if #t 5 (/ 2 0)) (Racket)
2. (let ((f (λ (x y) x))) (f 5 (/ 2 0))) (Racket)
3. let f x y = x in f 5 (div 2 0) (Haskell)
4. X = 2 / 0, Y = X. (Prolog)

*Soluție:*

1. Nu este eroare → `if` este funcție nestrictă.
2. Eroare, din cauza evaluării aplicative.
3. Nu este eroare, datorită evaluării leneșe (`y` nu este folosit).
4. Nu este eroare, `=` nu evluează calculele aritmetice.

# Examen PP – Seria CC — NOT EXAM MODE

16.06.2017

Timp de lucru 2 ore . 100p necesare pentru nota maximă

---

1. Determinați forma normală pentru următoarea expresie, ilustrând pașii de reducere:  
 $((\lambda x.\lambda y.\lambda z.(x\ y)\ \lambda x.x)\ z)$

*Soluție:*

$$((\lambda \underline{x}.\lambda y.\lambda z.(\underline{x}\ y)\ \lambda x.x)\ z) \rightarrow_{\beta} (\lambda \underline{y}.\lambda z.(\lambda x.x\ \underline{y})\ z) \rightarrow_{\alpha} (\lambda \underline{y}.\lambda w.(\lambda x.x\ \underline{y})\ z) \rightarrow_{\beta} \lambda w.(\lambda \underline{x}.x\ z) \rightarrow_{\beta} \lambda w.z$$

15p

2. Este vreo diferență (ca efect, la execuție) între cele două linii de cod Racket? Dacă da, care este diferența?; dacă nu, de ce nu diferă?

`(let ((a 1)) (let ((b a)) (+ a b)))`

`(let* ((a 1) (b a)) (+ a b))`

*Soluție:*

Nu este nicio diferență; `let*` este același lucru cu câte un `let` imbricat pentru fiecare definiție.

15p

3. Implementați în Racket funcția `f` care primește o listă și determină cel mai mare element. Folosiți, în mod obligatoriu, cel puțin o funcțională.

*Soluție:*

`(car (filter (λ(e) (null? (filter ((curry <) e) L))) L))`

sau

`(car (filter (λ(e) (null? (filter (λ(a) (< e a)) L))) L))`

sau

`(last (sort L <))`

15p

4. Sintetizați tipul funcției `f` (în Haskell): `f g h l = map (g . h) l`

*Soluție:*

`f :: a → b → c → d`

`map :: (e → i) → [e] → [i]`

`l :: c ⇒ c = [e]`

`(.) :: (t2 → t3) → (t1 → t2) → (t1 → t3)`

`a = g1 → g2 = t2 → t3`

`b = h1 → h2 = t1 → t2`

`g . h :: e → i = t1 → t3`

`c = [e] = [t1] = [h1]`

`d = [i] = [t3] = [g2]`

`f :: (t2 → t3) → (t1 → t2) → [t1] → [t3]`

15p

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste

un alt tip v, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.

Instantiați această clasă pentru tipul listă Haskell.

*Soluție:*

```
class Ended t where frontEnd :: t v -> v; backEnd :: t v -> v
instance Ended [] [Pair] [NestedL] [Triple] where
    frontEnd = head
    backEnd = last sau head . reverse
```

15p

6. Știind că *Cine spune multe, spune mai puțin decât cine tace*, și că *spune\_multe(Ion)* și *tace(Marcu)*, demonstrați folosind rezoluția că *spune\_mai\_putin(Ion, Marcu)* este adevărat.

*Soluție:*

$$\begin{aligned} &\forall x \forall y. spune\_multe(x) \wedge tace(y) \rightarrow spune\_mai\_putine(x, y) \\ &\neg spune\_multe(x) \vee \neg tace(y) \vee spune\_mai\_putine(x, y) \quad (1) \\ &spune\_multe(Ion) \quad (2) \\ &tace(Marcu) \quad (3) \\ &\neg spune\_mai\_putine(Ion, Marcu) \quad (4) \\ (1) + (2) \{x \leftarrow Ion\} &\Rightarrow \neg tace(y) \vee spune\_mai\_putine(Ion, y) \\ + (3) \{y \leftarrow Marcu\} &\Rightarrow spune\_mai\_putine(Ion, Marcu) \\ + (4) &\Rightarrow \text{clauza vidă} \end{aligned}$$

15p

7. Implementați în Prolog predicatul `x(L, A, B, N)` care determină, pentru o listă L, numărul N de elemente care sunt mai mari decât A și mai mici decât B. Nu folosiți recursivitate explicită.

*Soluție:*

```
x(L, A, B, N) :- findall(X, (member(X, L), X > A, X < B), S), length(S, N).
```

15p

8. Implementați un algoritm Markov care primește în sirul de intrare un număr binar și adună 1 la acest număr. Exemple:  $0 + 1 = 1$ ;  $1 + 1 = 10$ ;  $11 + 1 = 100$ ;  $100 + 1 = 101$

*Soluție:*

```
ag → ga
a → b
0b → 1c
1b → b0
b → 1c
c → .
→ a
```

15p

A

# Examen PP – Seria CC — NOT EXAM MODE

16.06.2017

Timp de lucru 2 ore . 100p necesare pentru nota maximă

---

1. Determinați forma normală pentru următoarea expresie, ilustrând pașii de reducere:  
 $((\lambda x.\lambda y.\lambda z.(y\ x)\ y)\ \lambda z.z)$

*Soluție:*

$$((\lambda \underline{x}.\lambda y.\lambda z.(y\ \underline{x})\ y)\ \lambda z.z) \rightarrow_{\alpha} ((\lambda \underline{x}.\lambda w.\lambda z.(w\ \underline{x})\ y)\ \lambda z.z) \rightarrow_{\beta} (\lambda \underline{w}.\lambda z.(\underline{w}\ y)\ \lambda z.z) \rightarrow_{\beta} \lambda z.(\lambda \underline{z}.\underline{z}\ y) \rightarrow_{\beta} \lambda z.y$$

15p

2. Este vreo diferență (ca efect, la execuție) între cele două linii de cod Racket? Dacă da, care este diferența?; dacă nu, de ce nu diferă?

```
(define a 2) (let ((a 1) (b a)) (+ a b))
(define a 2) (letrec ((a 1) (b a)) (+ a b))
```

*Soluție:*

În prima linie, definiția `(a 1)` este vizibilă în corpul `let`-ului, dar nu și în definiția lui `b`, care vede încă `a=2`; prima linie dă 3, a două dă 2.

15p

3. Implementați în Racket funcția `f` care primește o listă și determină cel mai mic element. Folosiți, în mod obligatoriu, cel puțin o funcțională.

*Soluție:*

```
(car (filter (λ(e) (null? (filter ((curry >) e) L))) L))
sau
(car (filter (λ(e) (null? (filter (λ(a) (> e a)) L))) L))
sau
(last (sort L >))
```

15p

4. Sintetizați tipul funcției `f` (în Haskell): `f x y z g = filter g [x, y, z]`

*Soluție:*

```
f :: a → b → c → d → e
d = g1 → g2
filter :: (t1 → Bool) → [t1] → [t1]
a = b = c (parte din aceeași listă)
t1 = a = b = c
e = [t1]
f :: t1 → t1 → t1 → (t1 → Bool) → [t1]
```

15p

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.

Instantiați această clasă pentru tipul `data Pair a = MakePair a a`

*Soluție:*

```
class Ended t where frontEnd :: t v -> v; backEnd :: t v -> v
instance Ended ] [Pair] [NestedL] [Triple] where
    frontEnd (MakePair x) = x
    backEnd (MakePair x) = x
```

15p

6. Știind că *Un om cum își așterne, aşa doarme*, și că *asterne(Nectarie, bine)* și *om(Nectarie)*, demonstrați folosind rezoluția că *doarme(Nectarie, bine)* este adevărat .

*Soluție:*

$$\begin{aligned} &\forall x \forall y. om(x) \wedge asterne(x, y) \rightarrow doarme(x, y) \\ &\neg om(x) \vee \neg asterne(x, y) \vee doarme(x, y) \\ &+ \neg doarme(Nectarie, bine) \{x \leftarrow Nectarie, y \leftarrow bine\} \Rightarrow \\ &\neg om(Nectarie) \vee \neg asterne(Nectarie, bine) \\ &+ om(Nectarie) \Rightarrow \neg asterne(Nectarie, bine) \\ &+ asterne(Nectarie, bine) \Rightarrow \text{clauza vidă} \end{aligned}$$

15p

7. Implementați în Prolog predicatul *x(L, A, B, N)* care determină, pentru o listă *L*, numărul *N* de elemente care nu sunt mai mari decât *A* și mai mici decât *B*. Nu folosiți recursivitate explicită.

*Soluție:*

```
x(L, A, B, N) :- findall(X, (member(X, L), (X < A; X > B)), S), length(S, N).
```

15p

8. Implementați un algoritm Markov care primește în sirul de intrare un număr binar și scade 1 din acest număr. Exemple:  $1 - 1 = 0$ ;  $10 - 1 = 1$ ;  $11 - 1 = 10$ ;  $100 - 1 = 11$ ;  $1010 - 1 = 1001$ . Este ok dacă numărul rezultat începe cu 0.

*Soluție:*

```
ag → ga
a → b
1b → 0c
0b → b1
c → .
→ a
```

15p

# Examen PP – Seria CC — NOT EXAM MODE

16.06.2017

Timp de lucru 2 ore . 100p necesare pentru nota maximă

---

1. Determinați forma normală pentru următoarea expresie, ilustrând pașii de reducere:

$$((\lambda x.\lambda y.\lambda z.(x\ y)\ \lambda x.y)\ a)$$

*Soluție:*

$$\begin{aligned} ((\lambda \underline{x}.\lambda y.\lambda z.(\underline{x}\ y)\ \lambda x.y)\ a) &\rightarrow_{\alpha} ((\lambda \underline{x}.\lambda w.\lambda z.(\underline{x}\ w)\ \lambda x.y)\ a) \rightarrow_{\beta} (\lambda \underline{w}.\lambda z.(\lambda x.y\ \underline{w})\ a) \rightarrow_{\beta} \\ &\lambda z.(\lambda x.y\ a) \rightarrow_{\beta} \lambda z.y \end{aligned}$$

15p

2. Este vreo diferență (ca efect, la execuție) între cele două linii de cod Racket? Dacă da, care este diferența?; dacă nu, de ce nu diferă?

```
(define a 2) (let ((c 2)) (let ((a 1) (b a)) (+ a b)))  
(define a 2) (let* ((c 2) (a 1) (b a)) (+ a b))
```

*Soluție:*

În prima linie, definiția (a 1) este vizibilă în corpul let-ului, dar nu și în definiția lui b, care vede încă a=2; prima linie dă 3, a două dă 2.

15p

3. Implementați în Racket funcția f care primește o listă și determină elementul cu cel mai mare modul. Folosiți, în mod obligatoriu, cel puțin o funcțională.

*Soluție:*

```
(car (filter (λ(e) (null? (filter (compose ((curry <) (abs e)) abs) L))) L))  
sau  
(car (filter (λ(e) (null? (filter (λ(a) (< (abs e) (abs a))) L))) L))  
sau  
(let ((M (last (sort (map abs L) <)))) (if (member M L) M (- 0 M)))
```

15p

4. Sintetizați tipul funcției f (în Haskell):  $f\ x\ y\ z\ g = map\ g\ [x,\ y,\ z]$

*Soluție:*

```
f :: a → b → c → d → e  
d = g1 → g2  
map :: (t1 → t2) → [t1] → [t2]  
a = b = c (parte din aceeași listă)  
t1 = a = b = c  
e = [t2]  
f :: t1 → t1 → t1 → (t1 → t2) → [t2]
```

15p

5. Scrieți definiția în Haskell a clasei Ended care, pentru un tip colecție t construit peste un alt tip v, definește o funcție frontEnd care extrage primul element din colecție și o funcție backEnd care extrage ultimul element din colecție.

Instantiați această clasă pentru tipul data NestedL a = A a | L [NestedL a]

*Soluție:*

```
class Ended t where frontEnd :: t v -> v; backEnd :: t v -> v
instance Ended ] [Pair] [NestedL] [Triple] where
    frontEnd (A a) = a; frontEnd (L l) = frontEnd $ head l
    backEnd (A a) = a; backEnd (L l) = backEnd $ last l
```

15p

6. Știind că *Un bogat când moare, săracul fluieră*, și că *bogat(Bill)*, *sarac(Bob)*, și *moare(Bill)*, demonstrați folosind rezoluția că *fluieră(Bob)* este adevărat .

*Soluție:*

```
∀x.∀y.bogat(x) ∧ sarac(y) ∧ moare(x) → fluiera(y)
¬bogat(x) ∨ ¬sarac(y) ∨ ¬moare(x) ∨ fluiera(y)
+bogat(Bill){x ← Bill} ⇒
¬sarac(y) ∨ ¬moare(Bill) ∨ fluiera(y)
+¬fluiera(Bob){⇒ ¬sarac(Bob) ∨ ¬moare(Bill)}
+moare(Bill) ⇒ ¬sarac(Bob)
+sarac(Bob) ⇒ clauza vidă
```

15p

7. Implementați în Prolog predicatul *x(L, M)* care determină, pentru o listă *L, M*, maximul listei. Nu folosiți recursivitate explicită.

*Soluție:*

```
x(L, M) :- member(M, L), forall(member(E, L), X > E).
```

15p

8. Implementați un algoritm Markov care primește în sirul de intrare un număr binar și adună 2 la acest număr. Exemple:  $1 + 10 = 10$ ;  $10 + 10 = 100$ ;  $1000 + 10 = 1010$ ;  $101 + 10 = 111$ ;  $111 + 10 = 1001$

*Soluție:*

```
ag → ga
ga → bg
0b → 1c
1b → b0
b → 1c
c → .
→ a
```

15p

# Examen PP – Seria CC — NOT EXAM MODE

16.06.2017

Timp de lucru 2 ore . 100p necesare pentru nota maximă

---

1. Determinați forma normală pentru următoarea expresie, ilustrând pașii de reducere:  
 $((\lambda z.\lambda y.\lambda x.(y z) y) \lambda y.y)$

*Soluție:*

$$((\lambda z.\lambda y.\lambda x.(y z) y) \lambda y.y) \rightarrow_{\alpha} ((\lambda z.\lambda w.\lambda x.(w z) y) \lambda y.y) \rightarrow_{\beta} (\lambda w.\lambda x.(w y) \lambda y.y) \rightarrow_{\beta} \lambda x.(\lambda y.y y) \rightarrow_{\beta} \lambda x.y$$

15p

2. Este vreo diferență (ca efect, la execuție) între cele două linii de cod Racket? Dacă da, care este diferența?; dacă nu, de ce nu diferă?

```
(let ((a 1) (b 2)) (+ a b))
((lambda (a b) (+ a b)) 1 2)
```

*Soluție:*

Nu este nicio diferență; `(let ((aivi)) corp)` este echivalent cu `(lambda (ai) corp)` aplicat parametrilor v<sub>i</sub>

15p

3. Implementați în Racket funcția f care primește o listă și determină elementul mai mare decât modulul oricărui alt element. Folosiți, în mod obligatoriu, cel puțin o funcțională.

*Soluție:*

```
(car (filter (λ(e) (null? (filter (compose ((curry <) e) abs) L))) L))
sau
(car (filter (λ(e) (null? (filter (λ(a) (< e (abs a))) L))) L))
sau
(last (sort L <)))
```

15p

4. Sintetizați tipul funcției f (în Haskell): `f g h l1 l2 = filter (g . h) (l1 ++ l2)`

*Soluție:*

```
f :: a → b → c → d → e
filter :: (i → Bool) → [i] → [i]
l1 :: c și l2 :: d ⇒ c = d = [i]
(.) :: (t2 → t3) → (t1 → t2) → (t1 → t3)
a = g1 → g2 = t2 → t3
b = h1 → h2 = t1 → t2
g . h :: i → Bool = t1 → t3
c = d = [i] = [t1] = [h1]
[t3] = [g2] = Bool
f :: (t2 → Bool) → (t1 → t2) → [t1] → [t1] → [t1]
```

15p

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.

Instantiați această clasă pentru tipul `data Triple a = T a a a`

*Soluție:*

```
class Ended t where frontEnd :: t v -> v; backEnd :: t v -> v
instance Ended ] [Pair] [NestedL] [Triple] where
    frontEnd (Triple x) = x
    backEnd (Triple x) = x
```

15p

6. Știind că *Ucenicul vrea să învețe pe dascăl*, și că *ucenic(Luke)* și *dascal(Yoda)*, demonstrați folosind rezoluția că *vrea-să-învețe(Luke, Yoda)* este adevărat .

*Soluție:*

$$\begin{aligned} &\forall x \forall y. ucenic(x) \wedge dascal(y) \rightarrow vrea\_sa\_invete(x, y) \\ &\neg ucenic(x) \vee \neg dascal(y) \vee vrea\_sa\_invete(x, y) \\ &+ ucenic(Luke) \{x \leftarrow Luke\} \Rightarrow \neg dascal(y) \vee vrea\_sa\_invete(Luke, y) \\ &+ \neg vrea\_sa\_invete(Luke, Yoda) \{y \leftarrow Yoda\} \Rightarrow \neg dascal(Yoda) \\ &+ dascal(Yoda) \Rightarrow \text{clauza vidă} \end{aligned}$$

15p

7. Implementați în Prolog predicatul `x(L, M)` care determină, pentru o listă `L`, `M`, minimul listei. Nu folosiți recursivitate explicită.

*Soluție:*

```
x(L, M) :- member(M, L), forall(member(E, L), X < E).
```

15p

8. Implementați un algoritm Markov care primește în sirul de intrare un număr binar și scade 2 din acest număr. Exemple:  $10 - 10 = 0$ ;  $100 - 10 = 10$ ;  $111 - 10 = 101$ ;  $10001 - 10 = 111$ . Este ok dacă numărul rezultat începe cu 0.

*Solutie:*

```
ag → ga
ga → gb
1b → 0c
0b → b1
c → .
→ a
```

15p

# Examen PP – Seria 2CC — NOT EXAM MODE

29.05.2018

ATENTIE: Aveți 2 ore · 100p pentru nota maximă · **Justificați răspunsurile!**

---

1. Reduceți la forma normală următoarea expresie, ilustrând pașii de reducere:

$$\lambda x.\lambda y.((\lambda x.\lambda y.x(y\ x))(x\ y))$$

*Soluție:*

$$\begin{aligned}\lambda x.\lambda y.((\lambda x.\lambda y.x(y\ x))(x\ y)) &\rightarrow_{\alpha} \lambda x.\lambda y.((\lambda x.\lambda z.x(y\ x))(x\ y)) \rightarrow_{\beta} \\ \lambda x.\lambda y.(\lambda z.(y\ x)(x\ y)) &\rightarrow_{\beta} \lambda x.\lambda y.(y\ x)\end{aligned}$$

2. Care este diferența între următoarele două linii de cod Racket

```
(let ((a 1) (b 2)) (let ((b 3) (c (+ b 2))) (+ a b c)))
```

```
(let* ((a 1) (b 2)) (let* ((b 3) (c (+ b 2))) (+ a b c)))
```

*Soluție:*

În prima definiția (b 3) nu este vizibilă în legarea lui c; rezultatul este 8, iar în a doua linie rezultatul este 9.

3. Scrieți în Racket o funcție echivalentă cu `zip` din Haskell, știind că

`zip :: [a] -> [b] -> [(a, b)].` Folosiți cel puțin o funcțională.

*Soluție:*

```
(define (zip L1 L2) (map cons L1 L2))
```

4. Sintetizați tipul funcției `f` în Haskell: `f x y z = x y . z`

*Soluție:*

```
y :: t  
z :: a -> b  
x y :: b -> c  
x :: t -> b -> c  
f x y z :: a -> c  
f :: (t -> b -> c) -> t -> (a -> b) -> a -> c
```

5. Instantiați clasa `Show` pentru funcții Haskell care iau un argument numeric, astfel încât afișarea unei funcții `f` va produce afișarea rezultatelor aplicării funcției pe numerele de la 1 la 10. E.g. afișarea lui `(+1)` va produce: 234567891011.

*Soluție:*

```
-# LANGUAGE FlexibleInstances #- -- nu este cerut în rezolvarea din examen
instance (Enum a, Num a, Show b) => Show (a -> b) where
    show f = concatMap (show . f) [1..10]
-- Enum nu este cerut în rezolvarea din examen
```

6. Folosiți list comprehensions pentru a produce fluxul listelor formate din primii 5 multipli ai fiecărui număr natural:

```
[[1,2,3,4,5],[2,4,6,8,10],[3,6,9,12,15],[4,8,12,16,20]] .
```

*Soluție:*

```
[take 5 [m | m <- [n..], mod m n == 0] | n <- [1..]]
```

7. Folosiți rezoluția pentru a demonstra că dacă *Ion este om* și *orice om are o bicicletă* atunci este adevărat că *Ion are bicicletă sau Ion este bogat* (folosiți predicatele *om(X)*, *areBicicleta(X)* și *bogat(X)*).

*Soluție:*

Avem premisele: *om(ion)* și  $\forall x. \text{om}(x) \Rightarrow \text{areBicicleta}(x)$

Concluzia: *areBicicleta(ion)  $\vee$  bogat(ion)*

Clauzele:

- (a)  $\{om/ion)\}$
- (b)  $\{\neg om(x) \vee areBicicleta(x)\}$
- (c)  $\{\neg areBicicleta/ion\}$  (prima parte a concluziei negate)
- (d)  $\{\neg bogat/ion\}$  (a doua parte a concluziei negate)
- (b) + (c)  $\{x \leftarrow ion\} \rightarrow \neg om/ion(e)$
- (a) + (e)  $\rightarrow$  clauza vidă

8. Scrieți un predicat Prolog `diff(A, B, R)` care leagă R la diferența mulțimilor (repräsentate ca liste) A și B.

*Soluție:*

```
intersect(A, B, R) :- findall(X, (member(X, A), member(X, B)), R).
```

9. Dat fiind un sir de date binare, scrieți un algoritm Markov care plasează la sfârșitul sirului suma modulo 2 a bitilor din sir. Exemplu: 101010110000111

$\rightarrow 1010101100001110$ ; 100110110110 → 1001101101101; 100110110111 → 1001101101110

*Soluție:*

CheckSum; 0,1 g<sub>1</sub>

a<sub>g1</sub>0 → 0a<sub>g1</sub>

a<sub>01</sub> → 1a<sub>1</sub>

a<sub>11</sub> → 1a<sub>0</sub>

a → .

→ a<sub>0</sub>

10. Considerăm o structură de date de tip listă circulară, caracterizată de conținutul său și de un cursor intrinsec structurii, poziționat la orice moment pe un element al listei. Avem următoarele funcționalități:

- Structura va putea fi creată pe baza unei liste obișnuite L; la creare cursorul va fi inițial poziționat pe elementul care era primul element din L;
- Operația *get*, care întoarce elementul de la poziția unde este cursorul;
- Operația *next*, care avansează cursorul cu o poziție spre dreapta;

Exemplu: avem lista circulară C, construită pe baza listei 1,2,3,1,5. Astfel:

$get(C) = 1$      $get(next(next(next(next(C)))))) = 5$

$get(next(C)) = 2$      $get(next(next(next(next(next(C))))))) = 1$

Se cere implementarea în Racket, Haskell sau Prolog a celor 3 funcționalități: crearea listei circulare, operația *get* și operația *next*.

*Soluție:*

Exemplu în Haskell:

```
circular l = concat . repeat $ l -- desfășurăm lista originală
get = head -- elementul curent
next = tail -- avansăm cursorul
```

# Examen PP – Seria 2CC — NOT EXAM MODE

29.05.2018

ATENTIE: Aveți 2 ore · 100p pentru nota maximă · **Justificați răspunsurile!**

---

1. Reduceți la forma normală următoarea expresie, ilustrând pașii de reducere:

$$\lambda x.\lambda y.((\lambda x.\lambda y.y(x)y)(y x))$$

*Soluție:*

$$\lambda x.\lambda y.((\lambda x.\lambda y.y(x)y)(y x)) \rightarrow_{\beta} \lambda x.\lambda y.(\lambda y.y(y x)) \rightarrow_{\beta} \lambda x.\lambda y.(y x)$$

2. Care este diferența între următoarele două linii de cod Racket

```
(let* ((a 1) (b 2) (c (+ a 2))) (+ a b c))
((lambda (a b c) (+ a b c)) 1 2 (+ a 2))
```

*Soluție:*

În a doua linie a nu este vizibil la invocarea funcției  $\lambda$ ; prima linie dă 6, a doua dă eroare.

3. Scrieți în Racket o funcție echivalentă cu `unzip` din Haskell, știind că `unzip :: [(a, b)] -> ([a], [b])`. Folosiți cel puțin o funcțională.

*Soluție:*

```
(define (unzip L) (cons (map car L) (map cdr L)))
```

4. Sintetizați tipul funcției `f` în Haskell: `f x y z = x . y z`

*Soluție:*

```
z :: t
y :: t → a → b
y z :: a → b
x :: b → c
f x y z :: a → c
f :: (b → c) → (t → a → b) → t → a → c
```

5. Instanțiați clasa `Show` pentru funcții Haskell care iau un argument numeric, astfel încât afișarea unei funcții `f` va produce afișarea rezultatelor aplicării funcției pe numerele de la 1 la 10. E.g. afișarea lui `(+1)` va produce: 234567891011.

*Soluție:*

```
-# LANGUAGE FlexibleInstances #- -- nu este cerut în rezolvarea din examen
instance (Enum a, Num a, Show b) => Show (a -> b) where
    show f = concatMap (show . f) [1..10]
-- Enum nu este cerut în rezolvarea din examen
```

6. Folosiți list comprehensions pentru a produce fluxul listelor de divizori pentru numerele naturale: `[[1], [1, 2], [1, 3], [1, 2, 4], [1, 5], [1, 2, 3, 6] ...]`.

*Soluție:*

```
[[d | d <- [1..n], mod n d == 0] | n <- [1..]]
```

7. Folosiți rezoluția pentru a demonstra că dacă *George este țăran și orice țăran are o sapă* atunci este adevărat că *George este deștept sau George are o sapă* (folosiți predicatele `țăran(X)`, `areSapă(X)` și `deștept(X)`).

*Soluție:*

Avem premisele: `țăran(george)` și  $\forall x. \text{țăran}(x) \Rightarrow \text{areSapă}(x)$

Concluzia: `areSapă(george) ∨ deștept(george)`

Clauzele:

(a) `{țăran(george)}`

- (b)  $\{\neg\text{taran}(x) \vee \text{areSapă}(x)\}$   
 (c)  $\{\neg\text{areSapă}(\text{george})\}$  (prima parte a concluziei negate)  
 (d)  $\{\neg\text{deștept}(\text{george})\}$  (a doua parte a concluziei negate)  
 (b) + (c)  $\{x \leftarrow \text{george}\} \rightarrow \neg\text{taran}(\text{george})(e)$   
 (a) + (e)  $\rightarrow$  clauza vidă
8. Scrieți un predicat Prolog `intersect(A, B, R)` care leagă R la intersecția multimilor (reprezentate ca liste) A și B.  
*Soluție:*
- ```
diff(A, B, R) :- findall(X, (member(X, A), \+ member(X, B)), R).
```
9. Dat fiind un sir de date binare, scrieți un algoritm Markov care plasează la sfârșitul sirului suma modulo 2 a bitilor din sir. Exemplu: 101010110000111  
 $\rightarrow 1010101100001110; 100110110110 \rightarrow 1001101101101; 100110110111 \rightarrow 1001101101110$   
*Soluție:*  
 CheckSum; 0,1 g<sub>1</sub>  
 $a_1g_10 \rightarrow 0a_1g_1$   
 $a_01 \rightarrow 1a_1$   
 $a_11 \rightarrow 1a_0$   
 $a \rightarrow .$   
 $\rightarrow a_0$
10. Considerăm o structură de date de tip listă circulară, caracterizată de conținutul său și de un cursor intrinsec structurii, poziționat la orice moment pe un element al listei. Avem următoarele funcționalități:
- Structura va putea fi creată pe baza unei liste obișnuite L; la creare cursorul va fi inițial poziționat pe elementul care era ultimul element din L;
  - Operația *get*, care întoarce elementul de la poziția unde este cursorul;
  - Operația *prev*, care deplasează cursorul cu o poziție spre stânga;
- Exemplu: avem lista circulară C, construită pe baza listei 1,2,3,1,5. Astfel:
- |                    |                                            |
|--------------------|--------------------------------------------|
| $get(C) = 5$       | $get(prev(prev(prev(prev(C))))) = 1$       |
| $get(prev(C)) = 1$ | $get(prev(prev(prev(prev(prev(C)))))) = 5$ |
- Se cere implementarea în Racket, Haskell sau Prolog a celor 3 funcționalități: crearea listei circulare, operația *get* și operația *prev*.  
*Soluție:*  
 Exemplu în Prolog:
- ```
circular(L, LC) :- reverse(L, LC). % inversăm lista. Elementul curent este ultimul element din L
get(LC, Current) :- LC = [Current | _]. % elementul current
next(LC, LCNew) :- LC = [Current, Rest], append(Rest, [Current], LCNew). % rotim și mergem la următorul element, ca și cum în lista originală am fi mers la precedentul.
```

# Examen PP – Seria CD — NOT EXAM MODE

14.06.2017

Timp de lucru 2 ore . 100p necesare pentru nota maximă

---

- Determinați forma normală pentru următoarea expresie, ilustrând pașii de reducere:  
 $((\lambda x.\lambda y.\lambda z.y \Omega) \lambda x.(x z))$

*Soluție:*

$$((\lambda \underline{x}.\lambda y.\lambda z.y \Omega) \lambda x.(x z)) \rightarrow_{\beta} (\lambda \underline{y}.\lambda z.\underline{y} \lambda x.(x z)) \rightarrow_{\alpha} (\lambda \underline{y}.\lambda w.\underline{y} \lambda x.(x z)) \rightarrow_{\beta} \lambda w.\lambda x.(x z)$$

- Ce întoarce următorul cod Racket și ce reprezintă valoarea întoarsă?

```
(let f ((n 0) (m 5)) (if (< n m) (+ n (f (+ n 1) m)) 0))
```

*Soluție:*

10, adună numerele de la n la m (fără m).

- De câte ori se evaluează expresia (E 1) în codul Racket de mai jos?

```
(let ((proc (λ (x) (+ (force x) (force x))))) (proc (delay (E 1))))
```

Transformați codul dat pentru a folosi închideri.

Este vreo diferență în execuție?

*Soluție:*

o singură dată; (let ((proc (λ (x) (+ (x) (x))))) (proc (λ() (E 1))))  
se va evalua E de 2 ori.

- Implementați în Haskell o funcție care primește 3 liste infinite și le adună element cu element. E.g.  $f [1..] [4..] [7..] = [12, 15..]$

*Soluție:*

```
f = zipWith3 (\x y z -> x + y + z) sau  
f (h1:t1) (h2:t2) (h3:t3) = h1 + h2 + h3 : f t1 t2 t3
```

- Se dă clasa Haskell: `class Summable t where sum :: (Num a) => t a -> a,`  
cu semnificația că funcția din clasă adună toate elementele conținute în argument.

- Instanțiați clasa dată pentru tipul listă;
- Definiți un tip *listă imbricată de elemente de tip a*;
- Instanțiați clasa dată pentru tipul definit.

*Soluție:*

```
instance Summable [] where sum l = foldl (+) 0 l  
data SL a = A a | S [SL a] deriving (Eq, Show)  
instance Summable SL where sum a = case a of (A a) -> a; (S l) -> foldl (+)  
0 $ map sum l
```

- Știind că *Vulpea care doarme nu prinde găini*, și că *vulpe(Vicky), gaina(Gini)*, și *doarme(Vicky)* , demonstrați folosind rezoluția că *prinde(Vicky, Gini)* este fals .

*Soluție:*

$\forall x.\forall y.vulpe(x) \wedge gaina(y) \wedge doarme(x) \rightarrow \neg prinde(x, y)$   
sau  $\forall x.vulpe(x) \wedge doarme(x) \rightarrow \neg \exists y.gaina(y) \wedge prinde(x, y)$

$\neg vulpe(x) \vee \neg gaina(y) \vee \neg doarme(x) \vee prinde(x, y)$

$+vulpe(Vicky)\{x \leftarrow Vicky\} \Rightarrow \neg gaina(y) \vee \neg doarme(Vicky) \vee prinde(Vicky, y)$   
 $\neg prinde(Vicky, Gini)y \leftarrow Gini\} \Rightarrow \neg gaina(Gini) \vee \neg doarme(Vicky)$   
 $+doarme(Vicky) \Rightarrow \neg gaina(Gini)$   
 $+gaina(Gini) \Rightarrow$  clauza vidă

7. Implementați în Prolog un predicatul `p(L1, L2, L3)`, care primește în L1 și L2 două liste de lungimi egale și leagă L3 la o listă de perechi, fiecare pereche conținând elementele de la același index din L1 și L2 (similar `zip` din Haskell). Exemplu: este adevărat `p([1, 2, 3], [a, b, c], [(1, a), (2, b), (3, c)])`.

*Soluție:*

```
zip(A, B, C) :- findall((X, Y), (nth0(I, A, X), nth0(I, B, Y)), C).
```

8. Implementați un algoritm Markov care primește un sir format din simboluri 0 și 1 și verifică dacă sirul începe și se termină cu același simbol. Dacă simbolurile de la început și de la sfârșit sunt identice, scrie la sfârșitul sirului simbolurile “ok”, altfel nu schimbă sirul cu nimic. Încercați să scrieți algoritmul pentru un alfabet de bază A oarecare.

*Soluție:*

1. StartEnd(); A g<sub>1</sub>, g<sub>2</sub>
2. ag<sub>1</sub> → g<sub>1</sub>bg<sub>1</sub>
3. bg<sub>1</sub>g<sub>2</sub> → g<sub>2</sub>bg<sub>1</sub>
4. g<sub>1</sub>bg<sub>1</sub> → g<sub>1</sub>okb
5. g<sub>1</sub>bg<sub>2</sub> → g<sub>1</sub>b
6. b → .
7. → a

# Examen PP – Seria CD — NOT EXAM MODE

14.06.2017

Timp de lucru 2 ore . 100p necesare pentru nota maximă

---

- Determinați forma normală pentru următoarea expresie, ilustrând pașii de reducere:  
 $((\lambda x.\lambda y.\lambda z.x\ (\lambda y.z\ \Omega))\ a)$

*Soluție:*

$$((\lambda \underline{x}.\lambda y.\lambda z.\underline{x}\ (\lambda y.z\ \Omega))\ a) \rightarrow_{\alpha} ((\lambda \underline{x}.\lambda y.\lambda w.\underline{x}\ (\lambda y.z\ \Omega))\ a) \rightarrow_{\beta} (\lambda \underline{y}.\lambda w.(\lambda y.z\ \Omega)\ a) \rightarrow_{\beta} \lambda w.z$$

- Ce întoarce următorul cod Racket și ce reprezintă valoarea întoarsă?

```
(let f ((n 120) (m 5)) (if (zero? m) 1 (f (/ n m) (- m 1))))
```

*Soluție:*

1, împarte n succesiv la m, m-1, m-2... cât timp m > 0.

- De câte ori se evaluează expresia (F 1) în codul Racket de mai jos?

```
(let ((proc (λ (x) (* (x) (x))))) (proc (λ () (F 1))))
```

Transformați codul dat pentru a folosi promisiuni.

Este vreo diferență în execuție?

*Soluție:*

de două ori; (let ((proc (λ (x) (\* (force x) (force x))))) (proc (delay (F 1)))) se va evalua F o singură dată.

- Implementați în Haskell o funcție care primește 3 liste infinite și determină maximul dintre ele la fiecare index. E.g.  $f [1,6,2 \dots] [8,3,4 \dots] [5,10,3 \dots] = [8,10,4 \dots]$

*Soluție:*

```
f = zipWith3 (\x y z -> max x $ max y z) sau  
f (h1:t1) (h2:t2) (h3:t3) = max h1 (max h2 h3) : f t1 t2 t3
```

- Se dă clasa Haskell: `class Addable t where add :: (Num a) => t a -> t a -> t a,` cu semnificația că funcția din clasă realizează adunarea, element cu element, a elementelor din cele două valori date ca argumente.

- Instanțiați clasa dată pentru tipul listă este suficient să acoperiți cazul cu liste de lungimi egale;
- Definiți un tip *pereche de elemente de tip a, în care ambele elemente ale perechii sunt de același tip*;
- Instanțiați clasa dată pentru tipul definit.

*Soluție:*

```
instance Addable [] where add = zipWith (+)  
data P a = P a a deriving (Eq, Show)  
instance Addable P where add (P a b) (P c d) = P (a+c) (b+d)
```

- Știind că *Tâlhar pe tâlhar nu fură*, și că *talhar(Ben)* și *talhar(Don)*, demonstrați folosind rezoluția că *fură(Ben, Don)* este fals .

*Soluție:*

$\forall x. \forall y. talhar(x) \wedge talhar(y) \rightarrow \neg fura(x, y)$   
 $\neg talhar(x) \vee \neg talhar(y) \vee fura(x, y)$   
 $+ talhar(Ben)\{x \leftarrow Ben\} \Rightarrow \neg talhar(y) \vee fura(Ben, y)$   
 $+ talhar(Don)\{y \leftarrow Don\} \Rightarrow fura(Ben, Don)$   
 $+ \neg fura(Ben, Don) \Rightarrow$  clauza vidă

7. Implementați în Prolog un predicatul `p(L1, L2, L3)`, care primește în L3 o listă de perechi și pune în L1 elementele de pe prima poziție din fiecare pereche și în L2 elementele de pe a doua poziție din fiecare pereche (similar `unzip` din Haskell). Exemplu: este adevărat `p([1, 2, 3], [a, b, c], [(1, a), (2, b), (3, c)])`.

*Soluție:*

```
unzip(A, B, C) :- findall(X, member((X, _), C), A), findall(X, member(_, X), C), B).
```

8. Implementați un algoritm Markov care primește un sir format din simboluri 0 și 1 și verifică dacă sirul începe și se termină cu simboluri diferite. Dacă simbolurile de la început și de la sfârșit sunt diferite, scrie la sfârșitul sirului simbolurile “ok”, altfel nu schimbă sirul cu nimic. Încercați să scrieți algoritmul pentru un alfabet de bază A oarecare.

*Soluție:*

1. StartEnd(); A g<sub>1</sub>, g<sub>2</sub>
2. ag<sub>1</sub> → g<sub>1</sub>bg<sub>1</sub>
3. bg<sub>1</sub>g<sub>2</sub> → g<sub>2</sub>bg<sub>1</sub>
4. g<sub>1</sub>bg<sub>1</sub> → g<sub>1</sub>b
5. g<sub>1</sub>bg<sub>2</sub> → g<sub>1</sub>okb
6. b → .
7. → a

Numele și grupa	1	2	3	4	5	6	7	8	9	10
A										

1. Încercuiți aparițiile legate și subliniați aparițiile libere ale variabilelor din expresia  $(\lambda x. (\lambda y. (x \ y)) \ \lambda z. (y \ z)) \ x$ .
2. Pentru codul Racket de mai jos, scrieți evoluția pas cu pas a execuției lui  $(pow \ 2 \ 5)$ , și ilustrați stiva la fiecare pas.

```
(define (pow a n)
  (cond ((zero? n) 1)
        ((odd? n) (* a (pow a (- n 1)))))
        (else (pow (* a a) (/ n 2)))))

(pow 2 5) []
(* 2 (pow 2 4)) [*2]
(* 2 (pow 4 2)) [*2]
(* 2 (pow 16 1)) [*2]
(* 2 (* 16 (pow 16 0))) [*16, *2]
(* 2 (* 16 1)) [*16, *2]
(* 2 16) [*2]
32 []
```

3. În Racket, fără a folosi recursivitate explicită:

- a) Scrieți o funcție curry care primește un predicat p și o listă L și întoarce lista elementelor x din L pentru care x respectă și  $x/2$  (împărțire întreagă) nu respectă p.

```
(define (f p)
  (λ (L)
    (filter (λ (x) (and (p x) (not (p (quotient x 2)))))) L)))
```

- b) Folosiți funcția de mai sus, scriind minimul necesar, pentru a extrage elementele divizibile cu 2 dar nu cu 4 ale unei liste numbers.

```
((f even?) numbers)
```

4. Folosind interfața Racket pentru fluxuri, implementați fluxul (1), (1 1), (1 1), (1 1 1), (1 1 1), (1 1 1 1), ... (primul element o dată, apoi fiecare element de câte 2 ori). Apoi implementați același flux folosind promisiuni, ca și cum interfața nu ar exista.

```
(define f1
  (let iter ((a '(1)) (b '(1 1)))
    (stream-cons a (stream-cons b (iter (cons 1 a) (cons 1 b))))))

(define f2
  (let iter ((a '(1)) (b '(1 1)))
    (cons a (delay (cons b (delay (iter (cons 1 a) (cons 1 b))))))))
```

5. Sintetizați tipul funcției f (în Haskell):  $f \ x \ y = \text{iterate } \$ \ x \ . \ y$

```
x :: a, y :: b, iterate \$ x . y :: c, f :: a -> b -> c
iterate :: (d -> d) -> d -> [d]
=> x . y :: d -> d, c = d -> [d]
(.) :: (g -> h) -> (e -> g) -> (e -> h)
=> x :: g -> h, y :: e -> g, x . y :: e -> h
=> e = d, h = d, a = g -> d, b = d -> g
=> f :: (g -> d) -> (d -> g) -> d -> [d]
```

6. Folosind un list comprehension (5p), instanțiați clasa Ord pentru tipul Student de mai jos (fiecare student este definit prin nume și o listă de triplete de forma (nume\_materie, notă\_materie, credite\_materie)) astfel încât studenții să fie ordonați după totalul creditelor la materii la care au obținut notă de trecere.

```
data Student = Student String [(String, Int, Int)] deriving (Eq, Show)
```

```

instance Ord Student where
    Student _ g1 <= Student _ g2 = credits g1 <= credits g2
        where
            credits g = sum [ cred | (_, grade, cred) <- g, grade >= 5 ]

```

7. Traduceți în FOL propoziția: "Orice măță blândă ori doarme, ori îl zgârie pe vreun om rău."

$$\forall X \bullet (măță(X) \wedge blândă(X)) \Rightarrow (doarme(X) \vee \exists Y \bullet (om(Y) \wedge rău(Y) \wedge zgârie(X, Y)))$$

8. Adăugați implementarea predicatului one\_bubble (care va trece o dată prin listă interschimbând elementele vecine care sunt în ordine strict descrescătoare) la următorul cod Prolog pentru algoritm bubble-sort:

```

bubble_sort(L, L) :- one_bubble(L, L), !.
bubble_sort(L, S) :- one_bubble(L, Bubbled), bubble_sort(Bubbled, S).

```

```

one_bubble([], []).
one_bubble([X], [X]). 
one_bubble([X, Y|Rest], [X|Res]) :- X <= Y, !, one_bubble([Y|Rest], Res).
one_bubble([X, Y|Rest], [Y|Res]) :- one_bubble([X|Rest], Res).

```

9. Pentru implementarea de mai jos, spuneți în câte moduri (și care sunt acestea, din punct de vedere al legării variabilelor) se va satisface scopul **sel(Y, [X, 2, Y, 4], 2, [1, Z, Z, 4])**.

```

sel(X, [X|List], Y, [Y|List]). 
sel(X, [X0|XList], Y, [X0|YList]) :- sel(X, XList, Y, YList).

```

2 moduri: Y = Z, Z = 2, X = 1 ; X = 1, Z = 2 ; false.

10. Problemă (de rezolvat în Haskell)

a) Implementați un tip de date MList pentru liste ce pot conține întregi, caractere sau perechi de întregi și caractere.

b) Implementați o funcție: filter' :: Char -> MList -> MList care filtrează o MListă astfel:

Dacă primul parametru are valoarea:

- 'i', atunci filter' întoarce o MListă ce conține doar valorile întregi
- 'c', atunci filter' întoarce o MListă ce conține doar caracterele
- 'p', atunci filter' întoarce o MListă ce conține doar perechile

c) Implementați o funcție care primește o MListă și întoarce lista caracterelor conținute, dacă în MListă se află doar caractere, sau o valoare cu semnificație de eșec - altfel. Valoarea cu semnificație de eșec nu trebuie să se poată obține dintr-o MListă care conține doar caractere (de exemplu, [] nu indică un eșec cert, ci ar putea proveni dintr-o MListă goală).

```

data Val = I Int | C Char | P (Int,Char) deriving Show
data MList = M [Val]

filter' :: Char -> MList -> MList
filter' c (M l) = M $ filter (f c) l
    where f 'i' (I _) = True
          f 'c' (C _) = True
          f 'p' (P _) = True
          f _ _ = False

conv :: MList -> Maybe [Char]
conv (M ((C c):rest)) =
    case (conv (M rest)) of
        Just l -> Just (c:l)
        Nothing -> Nothing
conv (M []) = Just []
conv _ = Nothing

```

Numele și grupa	1	2	3	4	5	6	7	8	9	10
B										

1. Folosind evaluare normală, aduceți la forma normală expresia  $(\lambda x. (\lambda y. (x \ y) \ \lambda z. (y \ z)) \ x)$ .

$$E \rightarrow (\lambda y. (x \ y) \ \lambda z. (y \ z)) \rightarrow (x \ \lambda z. (y \ z))$$

2. Pentru codul Racket de mai jos, scrieți evoluția pas cu pas a execuției lui  $(\text{mul } 3 \ 3)$ , și ilustrați stiva la fiecare pas.

```
(define (mul x y)
  (cond ((zero? y) 0)
        ((odd? y) (+ x (mul x (- y 1))))
        (else (+ (mul x (/ y 2)) (mul x (/ y 2))))))
```

(mul 3 3)	[]
(+ 3 (mul 3 2))	[+3]
(+ 3 (+ (mul 3 1) (mul 3 1)))	[+[..], +3]
(+ 3 (+ (+ 3 (mul 3 0)) (mul 3 1)))	[+3, +[..], +3]
(+ 3 (+ (+ 3 0) (mul 3 1)))	[+3, +[..], +3]
(+ 3 (+ 3 (mul 3 1)))	[+3, +3]
(+ 3 (+ 3 (+ 3 (mul 3 0))))	[+3, +3, +3]
(+ 3 (+ 3 (+ 3 0)))	[+3, +3, +3]
(+ 3 (+ 3 3))	[+3, +3]
(+ 3 6)	[+3]
9	[]

3. În Racket, fără a folosi recursivitate explicită:

a) Scrieți o funcție care primește o funcție binară  $f$  și două liste de aceeași lungime și întoarce lista aplicărilor lui  $f$  asupra elementelor de pe aceeași poziție (ex: pentru  $+$ , '(1 2 3) și '(4 5 6) întoarce '(5 7 9)).

```
(define (fun f L1)
  (λ (L2)
    (map f L1 L2)))
```

b) Folosiți funcția de mai sus pentru a aduna '(1 2 3) la toate listele dintr-o listă de liste.

```
(map (fun + '(1 2 3)) L)
```

4. Implementați în Racket fluxul multiplilor unui număr  $n$ , folosind închideri funcționale, ca și cum interfața Racket pentru fluxuri nu ar exista. Apoi obțineți (nu altfel decât prin extragere din flux) lista primelor 3 elemente din acest flux (pentru  $n = 5$ ).

```
(define (stream n)
  (let iter ((a 0))
    (cons a (λ () (iter (+ a n))))))
(let ((fives (stream 5)))
  (list (car fives) (car ((cdr fives))) (car ((cdr ((cdr fives)))))))
```

5. Sintetizați tipul funcției  $f$  (în Haskell):  $f x = \text{filter } \$ x . x . x$

```
x :: a, filter \$ x . x . x :: b, f :: a -> b
filter :: (c -> Bool) -> [c] -> [c]
=> x . x . x :: c -> Bool, b = [c] -> [c]
(.) :: (e -> g) -> (d -> e) -> (d -> g)
=> x :: e -> g, x . x :: d -> e, x . x . x :: d -> g
=> c = d, g = Bool
=> d = e = g = c = Bool
=> f :: (Bool -> Bool) -> [Bool] -> [Bool]
```

6. Folosind o funcție implementată cu găzzi (5p), instanțiați clasa Eq pentru tipul Student de mai jos (un student este definit prin nume și o listă de perechi de forma (nume\_materie, notă\_materie)) astfel încât doi studenți sunt egali dacă au același tip de rezultat la "PP" (există 3 tipuri: nepromovat, promovat cu nota sub 10, promovat cu nota 10).

```
data Student = Student String [(String, Int)] deriving Show
```

```
instance Eq Student where
    Student _ g1 == Student _ g2 = getRange g1 == getRange g2
        where
            f grade
                | grade < 5   = 1
                | grade < 10 = 2
                | otherwise   = 3
    getRange g = [ f grade | ("PP", grade) <- g ]
```

7. Folosiți rezoluția pentru a demonstra că  $\{a \vee b, a \Rightarrow c, b \Rightarrow d\} \vdash c \vee d$ .

Premise: 1.  $\{a, b\}$ ; 2.  $\{\neg a, c\}$ ; 3.  $\{\neg b, d\}$

Concluzie negată: 4.  $\{\neg c\}$ ; 5.  $\{\neg d\}$

Demonstrație: 6.  $\{\neg a\}$  (2,4) 7.  $\{\neg b\}$  (3,5) 8.  $\{b\}$  (1,6) 9. {} (7,8)

8. Folosind cut (!) pentru evitarea calculelor inutile (5p), implementați în Prolog predicatul switchUntilZero(?List1, ?List2) astfel încât List2 are aceleași elemente cu List1, cu excepția că până la întâlnirea unui eventual element 0 toate a-urile se transformă în b-uri și b-urile în a-uri.

```
?- switchUntilZero([a,b,1,d,b,0,a,b], R). --- R = [b, a, 1, d, a, 0, a, b].
?- switchUntilZero(R, [a,b,c,d,b,a,b]). --- R = [b, a, c, d, a, b, a].
```

```
other(a,b). other(b,a).
switchUntilZero([], []).
switchUntilZero([0|Rest], [0|Rest]) :- !.
switchUntilZero([X|Rest], [Y|Res]) :- other(X,Y), !, switchUntilZero(Rest, Res).
switchUntilZero([X|Rest], [X|Res]) :- switchUntilZero(Rest, Res).
```

9. Pentru implementarea de mai jos, spuneți în câte moduri (și care sunt acestea, din punct de vedere al legării variabilelor) se va satisface **pred([1, 2, 3, X], S)**.

```
pred(L, S) :- append(_, B, L), append(S, _, B), S = [_,_|_].
```

6 moduri: S = [1, 2] ; S = [1, 2, 3] ; S = [1, 2, 3, X] ; S = [2, 3] ;
S = [2, 3, X] ; S = [3, X] ; false.

10. Problemă (de rezolvat în Haskell)

a) (10p) Definiți tipul de date polimorfic "coadă" de elemente de un tip oarecare. O coadă va fi reținută sub forma a 2 stive (una de in, în care adăugăm, și una de out, din care scoatem (când se golește, vărsăm stiva de in în stiva de out, astfel încât să se respecte în continuare principiul FIFO)).

b) (20p) Pentru tipul de mai sus, furnizați signaturile (4p) și implementarea (16p) următorilor operatori:

- isEmpty care întoarce True pentru coada goală, altfel False
- top care întoarce elementul de la începutul cozii
- del care scoate elementul de la începutul cozii (întoarce coada fără acesta)
- ins care introduce un nou element în coadă (evident, la sfârșit)

```
data Queue a = Q [a] [a] deriving Show
```

```
isEmpty :: (Queue a) -> Bool
isEmpty (Q [] []) = True
isEmpty _ = False
```

```
top :: (Queue a) -> a
top (Q xs []) = last xs
top (Q _ (x:_)) = x
```

```
del :: (Queue a) -> (Queue a)
del (Q xs []) = Q [] $ tail $ reverse xs
del (Q xs (y:ys)) = Q xs ys
```

```
ins :: a -> (Queue a) -> (Queue a)
ins x (Q xs ys) = Q (x:xs) ys
```