Precizări:

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

1. Dați un exemplu de λ-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, întrebuințând **funcționala** `foldl` și **fără** a utiliza recursivitate explicită. Soluțiile 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!

```
(define f
  (lambda (x)
    (lambda (y)
      (+ x y))))

(let* ([f-1 (f 1)]
       [y 2]
       [x ((f 1) (f-1 1))])
  (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. .

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

$\square$

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

```
(let ([f (lambda (x y)
           (lambda (z)
             (+ x y z)))])
  (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.

```
f :: a -> b
x :: c -> d
a = c -> d
c = a -> b
a = (a -> b) -> d
```

Tip infinit! □

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

Soluție. .

```
{-# LANGUAGE FlexibleInstances #-}
instance Show (t a -> b -> c) where
    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, diferiți de numărul însuși. Primele două numere perfecte sunt $6 = 1+2+3$ și $28 = 1+2+4+7+14$.

Soluție. .

```
divisors x = [ d | d <- [ 1 .. x `div` 2 ], x `mod` d == 0 ]
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 $da(expeditor, destinatar, obiect)$.

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

□

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

Soluție. .

```
merge([], L, L).
merge(L, [], L).
merge([H1 | T1], [H2 | T2], [H1 | L]) :-
    H1 =< H2, merge(T1, [H2 | T2], L), !.
merge([H1 | T1], [H2 | T2], [H2 | L]) :-
    merge([H1 | T1], T2, L).
```

□

11. Scrieți un algoritm Markov, care calculează funcția *succesor* pentru numere naturale. Numerele au reprezentare unară, sub forma 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. .

```
(deffacts facts (list a b b c c c a a b c c c c))

(defrule remove
    ?l <- (list $?pre ?x ?x $?post)
  =>
    (retract ?l)
    (assert (list $?pre ?x $?post)))
```

□