Examen PP/2CD/28.05.19 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

B

Numele și grupa	1	2	3	4	5	6	7	8	9	10

1. Folosind evaluare normală, aduceți la forma normală expresia `(λx.(λy.(x y) λz.(y z)) x)`.

   `E → (λy.(x y) λz.(y z)) → (x λz.(y z))`

2. Pentru codul Racket de mai jos, scrieți evoluția pas cu pas a execuției lui `(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 = 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 gărzi (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).

Examen PP/2CD/28.05.19 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

```
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 ∨ b, a ⇒ c, b ⇒ d} |= c ∨ d.

Premise: 1. {a, b}; 2. {¬a, c}; 3. {¬b, d}
Concluzie negată: 4. {¬c}; 5. {¬d}
Demonstrație: 6. {¬a} (2,4) 7. {¬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 scopul **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
```