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

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 `(λx.(λy.(x y) λz.(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), (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 = 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 mâță blândă ori doarme, ori îl zgârie pe vreun om rău."

∀X●((mâță(X) ∧ blândă(X)) ⇒ (doarme(X) ∨ ∃Y●(om(Y) ∧ rău(Y) ∧ 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 algoritmul 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
```