Examen PP/2CD/10.06.22 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. Dați exemplu de o λ-expresie care conține 2 variabile x și y, astfel încât x are 2 apariții legate și una liberă, iar y are o apariție legată și una liberă.

Soluție (una din multe posibile): ((λx.λy.x x) y)
Barem: câte 2.5p pentru: 2 apariții legate x, o apariție liberă x, o apariție legată y, o apariție liberă y

2. In Racket, folosind **recursivitate pe coadă**, implementați funcția n-times-n care primește un parametru n și întoarce o listă care conține de n ori valoarea n. (ex: (n-times-n 4) => '(4 4 4 4)). **(7p)**
Ilustrați parțial (primele 3 apeluri recursive pe coadă) evoluția pas cu pas pentru (n-times-n 10). În cazul în care folosiți o funcție ajutătoare (sau un named let), atunci veți ilustra evoluția acesteia (acestuia). **(3p)**

Soluție:
```
(define (n-times-n n)
  (let loop ((i n) (acc '()))
    (if (zero? i)
        acc
        (loop (- i 1) (cons n acc)))))
```
(n-times-n 4) =>
(loop 4 '()) =>
(loop 3 '(4)) =>
(loop 2 '(4 4)) …

Barem:
a) 2p - funcție ajutătoare sau named let cu parametri corespunzători
1p - condiție de oprire corectă
1p - întoarce acc pe cazul de bază
3p - apel recursiv cu parametri actualizați corect
b) câte 1p pentru fiecare apel

3. În Racket, **fără a folosi recursivitate explicită**, scrieți o funcție **curry** care primește o listă de funcții și o listă de argumente și întoarce lista rezultatelor aplicării fiecărei funcții din prima listă pe argumentul de pe poziția corespunzătoare în a doua listă. (prima funcție pe primul argument, a doua pe al doilea, etc.) **(7p)**
Apelați corespunzător această funcție pe o primă listă care conține funcțiile list, odd? și null?, respectiv o a doua listă care conține valorile '(1), 2 și '(3) **(2p)** și precizați ce se obține în urma apelului. **(1p)**

Soluție:
```
(define (applications funcs)
  (λ (args)
    (map (λ (f a) (f a)) funcs args)))

((applications (list list odd? null?)) '((1) 2 (3))) => (((1)) #f #f)
```

Barem:
a) 3p - forma curry pentru cei 2 parametri
2p - map sau fold cu argumente corespunzătoare (ca număr și semnificație)
2p - funcție care ia f și a din cele 2 liste și calculează (f a) (sau adaugă (f a) în acc, la fold)
b) 1p - apel corespunzător formei curry
0.5p - listele construite cu o metodă corectă (funcția list sau quote unde sunt doar valori numerice)
0.5p - conținut corect liste
c) 1p - rezultat corect

4. Folosind interfața Racket pentru fluxuri, implementați fluxul x, x^2, x^3, x^4 … etc. **(5p)**
Folosiți-vă de această implementare pentru a implementa fluxul de liste (2 3), (4 9), (8 27), (16 81) … etc. **(5p)**

Soluție (se consideră definit stream-zip-with):
```
(define (powers x)
  (letrec ((pows (stream-cons x (stream-map (λ (e) (* x e)) pows))))
    pows))
(define f
  (stream-zip-with list (powers 2) (powers 3)))
```

Examen PP/2CD/10.06.22 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

Barem:

a) pt generare explicită:
- 2p - parametri relevanți în recursivitate
- 1p - generare corectă element curent
- 2p - apel recursiv cu parametri actualizați corect

pt generare implicită:
- 1p - generare prim element
- 2p - funcțională apelată pe nr și tip corect de argumente
- 2p - funcția (lui map, zip, etc.) face ce trebuie

b) idem a (pt generare implicită, 3p funcționala cu argumente corecte + 2p funcția)

5. Implementați funcția de la exercițiul 3 în Haskell (cu sau fără recursivitate explicită). **(3p)**
Sintetizați tipul acestei funcții (nu sunt necesare explicații formale, dar sunt necesare explicații). **(5p)**
Apelați funcția pe o primă listă care conține funcțiile odd și null și o a doua listă care conține argumentele 2 și [3] **(1p)** și precizați ce se obține în urma apelului. **(1p)**

Soluție:

a) `applications funcs args = zipWith (\f a -> f a) funcs args`

b) (1) zipWith :: (a -> b -> c) -> [a] -> [b] -> [c] (tipul funcției folosită de zipWith corespunde tipurilor elementelor listelor)
(2) \f a -> f a :: (t1 -> t) -> t1 -> t (f se aplică pe a => are un input de tipul lui a, funcția mare întoarce (f a) deci același tip cu ce întoarce f)
din (1),(2) => a = (t1 -> t), b = t1, c = t
=> applications :: [t1 -> t] -> [t1] -> [t] (applications primește ultimele 2 argumente ale lui zipWith și întoarce ce întoarce zipWith)

c) applications [odd, null] [2, [3]] => eroare întrucât listele nu sunt omogene (conțin elemente de tipuri diferite)

Barem:

a) 1p - luarea unui f și unui a și realizarea calculului f a
2p - zipWith sau apel recursiv cu argumente corecte (cu 0.5p pt cazul de bază, dacă se rezolvă recursiv)

b) 2p - explicație tip listă de funcții
1p - explicație tip listă de argumente
1p - explicație tip listă de rezultate
1p - explicație coincidență tip între inputul funcțiilor și argumente, între outputul funcțiilor și rezultate

c) 1p - apel scris corect
1p - semnalare eroare (explicată corect sau neexplicată, se scad 0.5p pentru eroare explicată eronat)

6. Fie definițiile următoare ale tipurilor Card (pentru cărți de joc) și Value (pentru valoarea asociată unei cărți):

```
data Card = C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | J | Q | K | A deriving Show
data Value = Only Int | Either Int Int deriving Show
```

Definiți clasa Valuable caracterizată de o singură funcție value care primește o valoare a tipului membru al clasei și întoarce o valoare de tip Value asociată acesteia. **(3p)**

Adăugați tipul Card la clasa Valuable, astfel încât cărților mici (de la 2 la 9) să li se asocieze propria lor valoare, cărților 10, J, Q, K să li se asocieze valoarea 10, iar asului (A) să i se asocieze fie valoarea 1, fie valoarea 11. **(7p)**

Soluție:

```
class Valuable t where
      value :: t -> Value

instance Valuable Card where
      value  A = Either 1 11
      value C2 = Only 2
      value C3 = Only 3
      value C4 = Only 4
      value C5 = Only 5
      value C6 = Only 6
      value C7 = Only 7
      value C8 = Only 8
      value C9 = Only 9
      value  _ = Only 10
```

Examen PP/2CD/10.06.22 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

Barem:

a) 0.5p - șablon class Valuable … where
0.5p - folosire variabilă de tip în definiția clasei
2p - signatură funcția value (1p input, 1p output)
b) 1p - șablon instance Valuable Card where
2p - definire value pe cărți mici
2p - definire value pe 10, J, Q, K
2p - definire value pe A

7. Demonstrați că {a, a ⇒ b, b ⇒ c} ⊨ c în două moduri: prin rezoluție **(5p)**, respectiv cu tabele de adevăr **(5p)**.

Soluție:

a) Cele 3 premise în formă clauzală:
(1) {a}
(2) {¬a, b}
(3) {¬b, c}

Rezultate ale aplicării rezoluției:
(4) {b} (din 1, 2)
(5) {c} (din 3, 4)

Obs: se acceptă demonstrat așa sau prin reducere la absurd, introducând ¬c și ajungând la rezolventul vid.

b)

a	b	c	a ⇒ b	b ⇒ c
T	T	T	T	T
T	T	F	T	F
T	F	T	F	T
T	F	F	F	T
….				

(rândurile cu a = F oricum nu interesează pentru că nu satisfac premisele)

Rândul marcat este singurul care satisface premisele {a, a ⇒ b, b ⇒ c}, și se observă că acesta satisface și concluzia c.

Barem:

a) 2p - transformarea implicațiilor în forma clauzală
3p - secvența de rezoluții care duce la {c}
b) 3p - construirea tabelei de adevăr total sau parțial cu explicația de ce restul nu contează
1p - identificarea rândurilor care satisfac premisele
1p - observația că acestea satisfac și c

8. Implementați, în Prolog, predicatul flatten care primește o listă de atomi și/sau liste (cu orice nivel de imbricare) și întoarce o listă care conține toți atomii din lista inițială. (ex: flatten([a,[b,[c],[[d]],[],[e]]], F) => F = [a, b, c, d, e].)

Soluție:

```
flatten([], []).
flatten([[]|Rest], Res) :- !, flatten(Rest, Res).
flatten([[X|R]|Rest], Res) :- !, flatten([X|R], Aux1), flatten(Rest, Aux2), append(Aux1,
Aux2, Res).
flatten([X|Rest], [X|Res]) :- flatten(Rest, Res).
```

Barem:

1p - cazul de bază
2p - primul element listă vidă (patterns, apel recursiv)
4p - primul element listă nevidă (patterns, flatten, flatten, append)
2p - primul element atom (patterns, apel recursiv)
1p - reguli mutual exclusive

9. Se dă un graf descris prin fapte Prolog de tip nod/1 și arc/2. Folosind un **metapredicat**, determinați lista tuturor perechilor de 2 noduri distincte între care există un drum de lungime 3 (3 arce). Lista **nu va conține duplicate**. Drumul poate conține același nod de mai multe ori, doar nodul inițial și cel final trebuie să fie diferite.

Soluție: `setof((X,Y), Z^T^(arc(X,Z), arc(Z,T), arc(T,Y), X \= Y), L).`

Barem:

1p - metapredicat cu template, goal, bag

Examen PP/2CD/10.06.22 Timp: 2h Ex 1-9: câte 10p, Pb 10: 30p, pentru maxim sunt suficiente 100p/120p

1p - template pereche
3p - identificare drum de 3 arce (cu corespondența variabilelor între ele și cu cele din template)
1p - X \= Y
2p - lista nu conține duplicate (setof sau findall((X,Y), (nod(X), nod(Y), once((arc(X,Z), arc(Z,T), arc(T,Y), X \= Y))), L))
2p - rezultat sub formă de o singură listă, nu satisfaceri succesive ale unui scop (Z^T^ sau findall ca mai sus)

10. Vom folosi tipurile Card și Value definite la exercițiul 6 (puteți presupune că exercițiul 6 este deja rezolvat, și că există funcția value care asociază fiecărei cărți valoarea acesteia) pentru a modela un joc de Blackjack. Pentru aceasta, ne interesează doar colecțiile de cărți pentru care suma valorilor cărților este maxim 21.

a) Implementați funcția correctValue :: Value -> Maybe Value care primește o valoare de tip Value din care caută să păstreze doar întregii care nu depășesc numărul 21. (ex: correctValue (Either 12 22) => Just (Only 12)) **(10p)**

b) Implementați funcția handValue :: [Card] -> Value care primește o listă de cărți și calculează suma valorilor cărților din listă. Dacă în listă apar unul sau mai mulțiași (A), atunci rezultatul va fi de tip Either suma1 suma2, unde suma1 este calculată pentru cazul când toți așii valorează 1, iar suma2 este calculată pentru cazul când un as valorează 11 și ceilalți valorează 1. (Nu are sens ca mai mai mult de un as să valoreze 11, pentru că suma ar depăși 21). (ex: handValue [A,A,J,C9] => Either 21 31) **(15p)**

c) Implementați **point free** funcția correctHandValue :: [Card] -> Maybe Value care primește o listă de cărți și calculează suma valorilor cărților din listă, cu corecția de la punctul a) deja aplicată, astfel încât să rămână doar sumele utile. (ex: correctHandValue [A,A,J,C9] => Just (Only 21)) **(5p)**

Soluție (pt punctul 2 am adăugat și clasa Eq la deriving):

```
correctValue :: Value -> Maybe Value
correctValue (Only s)
      | s > 21 = Nothing
      | otherwise = Just (Only s)
correctValue (Either s1 s2)
      | s1 > 21 && s2 > 21 = Nothing
      | s1 > 21 = Just (Only s2)
      | s2 > 21 = Just (Only s1)
      | otherwise = Just (Either s1 s2)

handValue :: [Card] -> Value
handValue cards =
      let aces = length $ filter (== A) cards
          others = filter (/= A) cards
          valueOthers = sum [ v | c <- others, let Only v = value c ]
      in if aces == 0 then Only valueOthers
                      else Either (aces + valueOthers) (aces + 10 + valueOthers)

correctHandValue :: [Card] -> Maybe Value
correctHandValue = correctValue . handValue
```

Barem:

a) 2p - Only s, s > 21 (peste tot, cele 2p se împart: pattern 0.5, condiție 0.5,
2p - Only s, altfel retur 1 (din care 0.5 pt aplicare Just, acolo unde este cazul))
2p - Either, ambele > 21
2p - Either, una > 21
2p - Either, ambele bune

b) 3p - valoare mână goală (dacă se folosește pattern matching, punctele se împart similar cu a))
6p - valoare mână fărăași
6p - valoare mână cu ași

c) 5p – compunerea funcțiilor de mai sus (din care 2p ordinea corectă)