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.

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)**

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)**

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)**

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)**

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)**

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

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].)

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.

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)**