Examen PP – Seria 2CC — NOT EXAM MODE

29.05.2018

ATENȚIE: Aveți 2 ore · 100p pentru nota maximă · **Justificați** răspunsurile!

---

1. Reduceți la forma normală următoarea expresie, ilustrând pașii de reducere:
   $\lambda x.\lambda y.((\lambda x.\lambda y.y\ \ (x\ \ y))\ \ (y\ \ x))$
   Soluție:
   $\lambda x.\lambda y.((\lambda x.\lambda y.y\ \ (x\ \ y))\ \ (y\ \ x)) \rightarrow_\beta \lambda x.\lambda y.(\lambda y.y\ \ (y\ \ x)) \rightarrow_\beta \lambda x.\lambda y.(y\ \ x)$

2. Care este diferența între următoarele două linii de cod Racket
   ```
   (let* ((a 1) (b 2) (c (+ a 2))) (+ a b c))
   ((lambda (a b c) (+ a b c)) 1 2 (+ a 2))
   ```
   Soluție:
   În a doua linie `a` nu este vizibil la invocarea funcției λ; prima linie dă 6, a doua dă eroare.

3. Scrieți în Racket o funcție echivalentă cu `unzip` din Haskell, știind că `unzip :: [(a, b)] -> ([a], [b])`. Folosiți cel puțin o funcțională.
   Soluție:
   ```
   (define (unzip L) (cons (map car L) (map cdr L)))
   ```

4. Sintetizați tipul funcției `f` în Haskell: `f x y z = x . y z`
   Soluție:
   ```
   z :: t
   y :: t → a → b
   y z :: a → b
   x :: b → c
   f x y z :: a → c
   f :: (b → c) → (t → a → b) → t → a → c
   ```

5. Instanțiați clasa `Show` pentru funcții Haskell care iau un argument numeric, astfel încât afișarea unei funcții `f` va produce afișarea rezultatelor aplicării funcției pe numerele de la 1 la 10. E.g. afișarea lui `(+1)` va produce: `234567891011`.
   Soluție:
   ```
   -# LANGUAGE FlexibleInstances #- -- nu este cerut în rezolvarea din examen
   instance (Enum a, Num a, Show b) => Show (a -> b) where
            show f = concatMap (show .  f) [1..10]
   -- Enum nu este cerut în rezolvarea din examen
   ```

6. Folosiți list comprehensions pentru aproduce fluxul listelor de divizori pentru numerele naturale: `[[1], [1, 2], [1, 3], [1, 2, 4], [1, 5], [1, 2, 3, 6] ...]` .
   Soluție:
   ```
   [[d | d <- [1..n], mod n d == 0] | n <- [1..]]
   ```

7. Folosiți rezoluția pentru a demonstra că dacă *George este țăran* și *orice țăran are o sapă* atunci este adevărat că *George este deștept sau George are o sapă* (folosiți predicatele $țăran(X)$, $areSapă(X)$ și $deștept(X)$).
   Soluție:
   Avem premisele: $țăran(george)$ și $\forall x.țăran(x) \Rightarrow areSapă(x)$
   Concluzia: $areSapă(george) \vee deștept(george)$
   Clauzele:
   (a) $\{țăran(george)\}$

(b) $\{\neg ță ran(x) \vee areSapă(x)\}$
(c) $\{\neg areSapă(george)\}$ (prima parte a concluziei negate)
(d) $\{\neg deștept(george)\}$ (a doua parte a concluziei negate)
(b) + (c) $\{x \leftarrow george\} \rightarrow \neg ță ran(george)(e)$
(a) + (e) $\rightarrow$ clauza vidă

8. Scrieți un predicat Prolog `intersect(A, B, R)` care leagă R la intersecția mulțimilor (reprezentate ca liste) A și B.
Soluție:
```
diff(A, B, R) :- findall(X, (member(X, A), \+ member(X, B)), R).
```

9. Dat fiind un șir de date binare, scrieți un algoritm Markov care plasează la sfârșitul șirului suma modulo 2 a biților din șir. Exemple: `101010110000111` $\rightarrow$ `1010101100001110`; `100110110110` $\rightarrow$ `1001101101101`; `100110110111` $\rightarrow$ `1001101101110`
Soluție:
CheckSum; 0,1 g_1
$ag_1 0 \rightarrow 0ag_1$
a01 $\rightarrow$ 1a1
a11 $\rightarrow$ 1a0
a $\rightarrow$.
$\rightarrow$ a0

10. Considerăm o structură de date de tip listă circulară, caracterizată de conținutul său și de un cursor intrinsec structurii, poziționat la orice moment pe un element al listei. Avem următoarele funcționalități:
    - Structura va putea fi creată pe baza unei liste obișnuite L; la creare cursorul va fi inițial poziționat pe elementul care era ultimul element din L;
    - Operația *get*, care întoarce elementul de la poziția unde este cursorul;
    - Operația *prev*, care deplasează cursorul cu o poziție spre stânga;

    Exemplu: avem lista circulară C, construită pe baza listei 1,2,3,1,5. Astfel:
    $get(C) = 5$ $\quad$ $get(prev(prev(prev(prev(C))))) = 1$
    $get(prev(C)) = 1$ $\quad$ $get(prev(prev(prev(prev(prev(C)))))) = 5$
    Se cere implementarea în Racket, Haskell **sau** Prolog a celor 3 funcționalități: crearea listei circulare, operația *get* și operația *prev*.
    Soluție:
    Exemplu în Prolog:
```
circular(L, LC) :- reverse(L, LC).% inversăm lista.  Elementul curent este ultimul
element din L
get(LC, Current) :- LC = [Current | _].  % elementul current
next(LC, LCNew) :- LC = [Current, Rest], append(Rest, [Current], LCNew).  % rotim și
mergem la următorul element, ca și cum în lista originală am fi mers la precedentul.
```

B