Examen PP varianta A — NOT EXAM MODE

31.05.2019

ATENȚIE: Aveți 2 ore · 1-9: 10p; 10: 30p · 100p pentru nota maximă · **Justificați** răspunsurile!

---

1. Reduceți expresia lambda $E = (\lambda x.(x\ \ (\lambda y.z\ \ x))\ \ \lambda x.x)$
   Soluție:
   $\rightarrow (\lambda x.(x\ \ z)\ \ \lambda x.x) \rightarrow (\lambda x.x\ \ z) \rightarrow z$

2. Se dă următorul cod Racket:
```
(define computation (delay (+ 5 5)))
(* 5 5)
(define (f x) (cons x (force computation)))
(map f '(1 2 3 4))
```
   (a) De câte ori se realizează adunarea?
   (b) Prima evaluare a adunării se realizează înainte sau după înmulțire?
   (c) Rescrieți codul pentru `computation` și pentru `f` folosind închideri funcționale în loc de promisiuni și răspundeți din nou la întrebările de la (a) și (b).

   Soluție:
   (a) o singură dată, la prima evaluare a lui `computation`.
   (b) după înmulțire, atunci când se apelează prima oară `(force computation)`
   (c)
```
(define computation (λ () (+ 5 5)))
(* 5 5)
(define (f x) (cons x (computation)))
(map f '(1 2 3 4))
```
   acum se apelează de 4 ori, la fiecare evaluare a lui `computation`; dar prima dată tot după înmulțire.

3. Date fiind două liste de numere `L1` și `L2`, scrieți în Racket codul care produce o listă de perechi `(x . n)`, unde `x` este un element din `L1`, iar `n` este numărul de apariții ale lui `x` în `L2`. E.g. pentru `L1 = (1 4 5 3)` și `L2 = (1 3 2 4 1 5 3 9)` rezultatul este `((1 . 2) (4 . 1) (5 . 1) (3 . 2))`. Nu folosiți recursivitate explicită.
   Soluție:
```
(map (lambda (x) (cons x (length (filter ((curry equal?) x) L)))) '(1 4 5 3))
```
   sau
```
(map (lambda (x) (cons x (length (filter (lambda (y) (equal? x y)) L)))) '(1 4 5 3))
```

4. Sintetizați tipul următoarei funcții în Haskell: `f x y = x y (y x)`
   Soluție:
   `x :: a -> b -> c`
   `y :: a` , dar și `y :: (a -> b -> c) -> b` $\Rightarrow$ `a = (a -> b -> c) -> b` Eroare, deoarece `a` nu poate unifica cu o expresie de tip care îl conține strict pe `a`.

5. (a) Câți pași de concatenare sunt realizați pentru evaluarea expresiei Racket `(car (append '(1 2) '(3 4)))` ?
   (b) Dar pentru expresia Haskell `head $ [1, 2] ++ [3, 4]` ?

   Soluție:
   (a) Se concatenează întregime listele, deci doi pași.
   (b) Este suficient un singur pas pentru ca `head` să întoarcă primul element.

6. Evidențiați o posibilă instanță a clasei Haskell de mai jos:
```
class MyClass c where
  f :: c a -> a
```

Soluție:

```
instance MyClass [] where
  f = head
```

7. Transformați propoziția „Nu tot ce zboară se mănâncă.” în logică cu predicate de ordinul întâi.
   Soluție:
   $\exists x.zboara(x) \wedge \neg se\_mananca(x)$ – există și lucruri care zboară și nu se mănâncă
   sau
   $\neg(\forall x.zboara(x) \Rightarrow se\_mananca(x))$ – nu este adevărat că orice care zboară automat se și mănâncă

8. Se dă programul Prolog:
   ```
   p(R, S) :- member(X, R),
           findall(Y, (member(Y, R), Y \= X), T), !, q(X, T, S).
   q(X, A, [X|A]). q(X, [A|B], [A|C]) :- q(X, B, C).
   ```
   Dacă predicatul `p` primește în primul argument o listă, la ce valori leagă al doilea argument? Câte soluții are interogarea `p([1, 2, 3, 4], S)` ?
   Soluție:
   Ia primul element (și elimină duplicatele lui) și îl pune pe diverse poziții ale listei, inclusiv pe prima. Patru soluții: `[1, 2, 3, 4]`, `[2, 1, 3, 4]`, `[2, 3, 1, 4]`, `[2, 3, 4, 1]`

9. Se dau următoarele relații genealogice prin predicatul `c(Parinte, Copil)`. Implementați predicatul `frati(X, F)`, care leagă `F` la lista de frați ai lui `X` (dacă există). De exemplu, pentru definițiile de mai jos, interogarea `frati(herodot, F)` leagă `F` la `[faramir, george]`.
   ```
   c(alex, celia).  c(alex, delia).  c(alex, marcel).
   c(barbra, celia).  c(barbra, delia).  c(barbra, marcel).
   c(delia, faramir).  c(delia, george).  c(delia, herodot).
   c(erus, faramir).  c(erus, george).  c(erus, herodot).
   ```
   Soluție:
   ```
   frati(X, F) :- c(P, X), !, findall(Y, (c(P, Y), Y \= X), F). sau
   frati(X, F) :- findall(Y, (c(P, X), c(P, Y), Y \= X), F1), sort(F1, F).
   ```

10. PROBLEMA (Poate fi implementată în orice limbaj studiat la PP.) Se urmărește implementarea unui *multi-map*, care este un tabel asociativ în care unei chei i se pot asocia oricâte valori.
    (a) Descrieți reprezentarea *multi-map*-ului. Pentru Haskell, dați definiția tipului de date polimorfic. Definiți funcția/predicatul `lookup'`, care extrage lista tuturor valorilor asociate cu o cheie.
    (b) Definiți funcția/predicatul `insert'`, pentru adăugarea unei noi asocieri între o cheie și o valoare.
    (c) Definiți funcția/predicatul `map'`, care aplică o funcție/predicat pe fiecare valoare din *multi-map*. NOTĂ: în Prolog, `map'` va aplica întotdeauna un același predicat `p(+VIn, -VOut)`.
    Soluție:
    Racket:
    ```
    (define multimapExample '((a 1 2 3) (b 5 6 7) (c 7 8)))

    (define (lookup k m) (cdr (assoc k m)))
    (lookup 'b multimapExample)

    (define (insert k v m)
    (let-values (((bef aft) (splitf-at m (λ(kv) (not (equal?  (car kv) k))))))
    (if (null?  aft)
      (cons (list k v) m)
      (append bef (list (cons k (cons v (cdar aft)))) (cdr aft))
      )))
    (insert 'd 5 multimapExample)
    (insert 'b 9 multimapExample)
    ```

```
(define (mmap f m)
  (map
    (lambda (kv) (cons (car kv) (map f (cdr kv))))
  m))
  (mmap add1 multimapExample)
```

Haskell:

```
data MultiMap k a = MM [(k, [a])] deriving (Eq, Show)

ins ::  Eq k => k -> a -> MultiMap k a -> MultiMap k a
ins k a (MM lst) = MM $ case back of
  [] -> (k, [a]) :  front
  (_, as) :  back -> (k, a :  as) :  front ++ back
  where
  (front, back) = break ((== k) .  fst) lst

  map' ::  (a -> b) -> MultiMap k a -> MultiMap k b
  map' f (MM lst) = MM $ map (\(k, as) -> (k, map f as)) lst

  test = ins 2 'z' $ ins 3 'c' $ ins 2 'b' $ ins 1 'a' $ MM []
```

Prolog:

```
lookup(K, MM, Values) :- member((K, Values), MM).

insert(K, V, MM, Out) :-
  select((K, L), MM, MM1), !,
  Out = [(K, [V|L]) | MM1].
insert(K, V, MM, [(K, [V]) | MM]).

f(V, V1) :- V1 is V + 1.

map(MM, Out) :-
  findall((K, L1),
    ( member((K, L), MM),
      findall(E1, (member(E, L), f(E, E1)), L1)),
    Out).
```