Nume: _____ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 | 10 | —

NOT EXAM MODE

Examen PP CC - var A 21.06.2024 | | | | | | | | | | | | |

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

---

1. Pentru următoarea λ-expresie: (a) câți beta-redecși există în expresie? (b) ce variabile *distincte* există în expresie?: $(\lambda x.(x\ \ x)\ \ (\lambda x.x\ \ y))$
   Soluție:
   2 β-redecși, o variabilă x în primul β-redex, o variabilă x și o variabilă y în al doilea.

2. Scrieți în Racket o funcția `serialChecker` care primește o listă de perechi, fiecare pereche având pe prima poziție o funcție și pe a doua poziție o listă. `serialChecker` verifică pentru fiecare pereche dacă toate elementele listei sunt validate de funcția din pereche. Exemplu: `(serialChecker (list (cons odd? '(1 3 5)) (cons even? '(2 3 4)) (cons null? '(() () ())) (cons zero? '(1 2 3))))` → `'(#t #f #t #f)` În implementare folosiți **cel puțin o funcție explicit recursivă, ce folosește recursivitate pe coadă**. Indicați care este acea funcție.
   Soluție:
   ```
   (define (serialCheckerStack L)
     (if (null? L) null
       (cons (let serialCheckerTail ((values (cdar L)))
         (if (null? values) #t
           (and ((caar L) (car values)) (serialCheckerTail (cdr values)))
           )) (serialCheckerStack (cdr L)))))
   ```

3. Implementați funcția de la exercițiul anterior **fără a folosi recursivitate explicită**.
   Soluție:
   ```
   (define (serialChecker L) (map (λ (p) (andmap (car p) (cdr p))) L))
   ```

4. Construiți în Racket sau în Haskell, folosind o definiție **implicită**, fluxul 1+0, 1-1, 0+2, 2-3, -1+4, 3-5, . . . , adică 1, 0, 2, -1, 3, -2, . . . , în care fiecare element este elementul anterior din care se adună sau scade, alternativ, indexul elementului în flux.
   Soluție:
   ```
   alternating = 1 :  zipWith (+) alternating (zipWith (*) nat (map ((-1)**) nat))
   ```

5. Sintetizați, **ilustrând** procesul de sinteză, tipul funcției Haskell `f g = map (\a -> g . a)`
   Soluție:
   map :: (t → b) → [t] → [b]
   a :: t
   g . a :: b

   (.) :: (d → e) → (c → d) → c → e
   deci
   g :: d → e
   a :: c → d
   t = c → d
   b = c → e

   ⇒
   f :: (d → e) → [c → d] → [c → e]

6. Instanțiați clasa Num (doar operatorii de adunare și înmulțire) pentru tipul listă Haskell, unde 'adunarea' a două liste rezultă în concatenarea lor, iar 'înmulțirea' a două liste rezultă în adunarea lor element cu element.
   Soluție:

```
instance Num a => Num [a] where
  (+) = (++)
  (*) = zipWith (+)
```

7. Știind că „Nu este pentru cine se potrivește, ci este pentru cine se nimerește”, și știind că $se\_potriveste(Marcel, Irina)$, demonstrați folosind metoda rezoluției că $\neg este\_pentru(Marcel, Irina)$.

   Soluție:
   Propoziția se reprezintă ca
   $se\_potriveste(x, y) \Rightarrow \neg este\_pentru(x, y) \ \wedge \ se\_nimerește(x, y) \Rightarrow este\_pentru(x, y)$ și se traduce în două clauze:
   $\neg se\_potriveste(x, y) \vee \neg este\_pentru(x, y)$ și
   $\neg se\_nimerește(x, y) \vee este\_pentru(x, y)$
   Prima clauză de mai sus rezolvă cu $se\_potriveste(Marcel, Irina)$, cu substituția $x \leftarrow Marcel, y \leftarrow Irina$, iar din pasul de rezoluție rezultă concluzia.

8. Implementați în Prolog predicatul `select(List, X, Result)` care produce în `Result` o listă identică cu `List`, cu excepția că în `Result` nu apare nicio apariție a elementului `X`. Implementați folosind recursivitate explicită.
   Soluție:
```
select([], _, []).
select([X|R], X, R1) :- !, select(R, X, R1).
select([H|R], X, [H|R1]) :- select(R, X, R1).
```

9. Implementați predicatul de la exercițiul anterior folosind **metapredicate**.
   Soluție:
```
select(L, X, Out) :- findall(A, (member(A, L), A \= X), Out).
```

10. Se consideră un tip de date, pe care îl numim *serie de valori*, care este o enumerare de înregistrări, fiecare înregistrare având o cheie și o valoare. Cheile dintr-o serie de valori sunt toate de același tip `t`. Valorile sunt toate numere întregi. De exemplu, putem avea seria de valori a:1, a:2, b:10, c:10, b:9, c:5, a:3.
    (a) definiți în Haskell tipul descris mai sus. Implementați funcția `selectSeries` care, pentru o cheie de tip `t` și o serie de valori peste tipul `t`, întoarce o listă formată din valorile asociate cu acea cheie, în ordinea în care au fost înregistrate. Pentru exemplu și cheia `a`, funcția întoarce lista `[1,2,3]`.
    (b) implementați funcția `valueCounts` care pentru o serie de valori întoarce o listă de perechi unde primul element este o cheie din serie, iar al doilea element este numărul de valori asociate cu acea cheie. Pentru exemplul de mai sus, funcția întoarce `[('a',3),('b',2),('c',2)]`.
    (c) implementați funcția `allMonotonic`, care pentru o serie de valori verifică că evoluția este monoton crescătoare sau descrescătoare *pentru fiecare cheie în parte*. Pentru exemplu, funcția întoarce `True`.
    Soluție:
```
data ValueSeries a = VS { list ::  [(a, Integer)] }
selectSeries ::  Eq a => a -> ValueSeries a -> [(a, Integer)]
selectSeries key = filter ((==key) .  fst) .  list
unique ::  Eq a => ValueSeries a -> [a]
unique = nub .  map fst .  list
 where
   nub [] = []
   nub (h:t) = h :  nub [x | x <- t, x /= h]
valueCounts vs = map (\key -> (key, length $ selectSeries key vs)) (unique vs)
allMonotonic vs = and $ map (\k -> isMon $ selectSeries k vs) $ unique vs
 where
   isMon values
     | length values == 1 = True
     | values !!  0 < values !!  1 = mon (<) values
     | otherwise = mon (>) values
   mon f [_] = True
```

```
mon f (h1:h2:t) = f h1 h2 && mon f (h2:t)
```