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

NOT EXAM MODE

Examen PP CB – var B 03.06.2025 | | | | | | | | | | | | |

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

---

1. (a) Reduceți la forma normală expresia de mai jos, ilustrând pașii de reducere sau justificând cu rigurozitate. (b) Ce variabile libere și ce variabile legate există în expresie?
   $((\lambda x.\lambda y.x\ \ ((\lambda x.\lambda y.y\ \ A)\ \ B))\ \ ((\lambda x.\lambda y.y\ \ C)\ \ D))$
   Soluție:
   Se aplică un selector care întoarce primul argument pe o expresie care aplică un selector care întoarce al doilea argument. Rezultatul este deci B.
   Sunt 10 variabile în expresie – câte 2 variabile x și y pentru fiecare selector și cele 4 variabile A, B, C, D.

2. Descrieți care este efectul următorului program Racket, și al evaluării expresiei Haskell de mai jos:
   1. `(define (A) (A)) (define (B) (A)) (define C 1) (define D 2)`
   2. `([lambda (x y) x] ([lambda (x y) y] (A) (B)) ([lambda (x y) y] C D))`

   ```
   let a = a; b = b; c = 1; d = 2 in (\x y -> x) ((\x y -> y) a b) ((\x y -> y) c d)
   ```
   Soluție:
   În codul Racket se evaluează toate argumentele, indiferent ce întorc funcțiile, deci nu se oprește din cauza lui `A`.
   Evaluarea expresiei Haskell nu se oprește din cauză că trebuie să evalueze pe b.

3. La ce se evaluează următoarea expresie în Racket?
   `(let* [(x 3) (y 4) (f (delay (λ (y) (+ x y))))]  (let [(x 1)] ((force f) x)))`
   Soluție:
   Delay capturează valoarea lui `x` din `let*`. Adunarea va fi cu `x` din `let`. Rezultatul este 4, din x=3 + x=1.

4. Construiți în Haskell fluxul x în care $x_0 = 2$, $x_1 = 0$, și $x_n = \sum_{i=0}^{n-2}\{x_i \text{ dacă } x_i \text{ par altfel } 0\}$, pentru $n \geq 2$. Fluxul este 2,0,2,2,4,6,10,16,26,42,68,110,... și, de exemplu, elementul 16, de pe poziția 7, este suma elementelor pare de pe pozițiile 0-5. **Evitați recursivitatea explicită**.
   Soluție:
   ```
   f = 2 :  map (\idx -> sum $ filter even $ take idx f) [0..]
   ```
   Nu este neapărat nevoie de verificarea de partitate pentru că toate elementele sunt pare.

5. Sintetizați, **ilustrând** procesul de sinteză, tipul funcției Haskell `f t = zipWith (.) . map t`
   Soluție:
   `zipWith :: (a → b → c) → [a] → [b] → [c]`
   `(.) :: (t2 → t3) → (t1 → t2) → (t1 → t3)` pentru `(.)` argumentul lui `zipWith`
   `a = t2 → t3`
   `b = t1 → t2`
   `c = t1 → t3`
   `zipWith (.) :: [t2 → t3] → [t1 → t2] → [t1 → t3]`
   `map :: (g → h) → [g] → [h]`
   `t = g → h` pentru că `t` este argumentul lui `map`, și atunci:
   `map t :: [g] -> [h]`
   `zipWith` este compus cu `map t`, și atunci
   `[h] = [t2 → t3]`
   `h = t2 → t3`
   ⇒ `f :: (g → t2 → t3) → [g] → [t1 → t2] → [t1 → t3]`

6. Instanțiați în Haskell clasa `Num` pentru triplete de valori (fiecare valoare de un tip arbitrar), implementând numai operatorul `+`, care să efecteze operația cerută numai pe prima valoare din fiecare triplet, în rest păstrând valorile din primul triplet.
   Soluție:
   ```
   instance Num a => Num (a, b, c) where
     (a1, b, c) + (a2, _, _) = (a1 + a2, b, c)
   ```

7. Știind că "Pe cine nu doare, nu câștigă", demonstrați prin metoda rezoluției că dacă *câștigă*(Ion), atunci *doare*(Ion).
Soluție:
Propoziții:
$\forall x.\neg doare(x) \rightarrow \neg castiga(x)$ cu FNC $doare(x) \vee \neg castiga(x)$ (1)
$castiga(Ion)$ (2)
Adăugăm concluzia negată: $\neg doare(Ion)$(3)
(1) + (3) cu $\{x \leftarrow Ion\} \Rightarrow \neg castiga(Ion)$
+ (2) $\rightarrow$ clauza vidă

8. Scrieți un predicat Prolog `up` care concatenează secvențele (cel puțin două elemente) strict crescătoare dintr-o listă. Exemplu:
`up([5, 1, 2, 3, 2, 3, 1, 1, 0, 9, 10], LS)` $\rightarrow$ `LS = [1, 2, 3, 2, 3, 0, 9, 10]`
Soluție:
```
up([], []).
up([H, H1 | T], [H, H1 | LS]) :- H1 > H, !, up(T, H1, LS).
up([_ | T], LS) :- up(T, LS).
up([], _, []) :- !.
up([H | T], E, [H | LS]) :- H > E, !, up(T, H, LS).
up(L, _, LS) :- up(L, LS).
```

9. Implementați în Prolog, folosind metapredicate, predicatul `getBest(+List, -BestValue)`, care primește în `List` o listă de perechi cheie – valoare, și, dat fiind un predicat `cond`, întoarce *cheia* corespunzătoare celei mai mici *valori* dintre perechile cu o *valoare* care respectă `cond`.
Soluție:
```
getBest(List, KM) :- findall((K, V), (member((K, V), List), cond(V)), L),
   member((KM, VM), L), forall(member((_, Vi), L), VM =< Vi).
```

10. Implementați cerințele următoare într-un limbaj la alegere dintre Haskell, Racket și Prolog, **evitând recursivitatea explicită**. Considerăm o serie de timp ca fiind o serie de valori reale, asociate cu momente de timp (valori întregi). De exemplu, putem avea seria `[20, 15, 16, 19, 18, 20, 18, 22, 22, 21, 20, 25]`, cu valori pentru 12 momente de timp. Putem *eticheta* fiecare valoare din seria de timp cu "unknown", "up" (mai mare decât precedenta), "down" (mai mică) sau "same" (la fel).
(a) În Haskell, definiți tipul (tipurile) de date, ca tip(uri) noi, necesare pentru a reprezenta o serie de timp, inclusiv cu suport pentru etichete. În celelalte limbaje, descrieți modalitatea de reprezentare. Implementați funcția/predicatul `create` care pentru o valoare `start` și o valoare `stop` construiește seria de timp cu valorile dintre momentele `start` și `stop`, cu etichete "unknown". Pentru a obține valoarea de la fiecare moment de timp, folosiți o funcție / un predicat `get` care primește un moment de timp și întoarce valoarea de la acel moment de timp. Considerăm `get` deja definit(ă).
(b) Implementați funcția/predicatul `label` care primește o serie de timp și întoarce o serie de timp cu etichete asociate cu fiecare valoare. Pentru primul element al seriei de tip se va asocia eticheta "unknown". Pentru exemplul de mai sus și seria de timp între momentele 2 și 6 va rezulta o serie
`2 - 16.0 - Unknown, 3 - 19.0 - Up, 4 - 18.0 - Down, 5 - 20.0 - Up, 6 - 18.0 - Down`
(c) Implementați funcțiile/predicatele `select`, care primește o etichetă și o serie de timp și selectează prima valoare și apoi valorile cu eticheta dată; și `deltas`, care primește o serie de timp și calculează diferențele dintre valori succesive. Petnru seria de la (b), `select` cu eticheta "up" va da seria de tip `2 - 16.0 - Unknown, 3 - 19.0 - Up, 5 - 20.0 - Up`, iar `deltas` peste această valoare va da lista `[3.0,1.0]`.
Soluție:
```
data Direction = Unknown | Up | Down | Same deriving (Show, Eq)
data TimeSeries = TS [(Integer, Float, Direction)] deriving (Show, Eq)

create ::  Integer -> Integer -> TimeSeries
create start stop = TS reverse foldl (time -> (time, get time, Unknown ) :  acc) [] [start..stop]

label ::  TimeSeries -> TimeSeries
label (TS triples) = TS reverse foldl
  ((t, val, _) -> case acc of
     -> [(time, value, Unknown)]
```

```
    (_, v2, _) :  _ ->
      (t, val, let symb | value > v2 = Up | value < v2 = Down | otherwise = Same in symb) :  acc
  ) [] triples

select ::  Direction -> TimeSeries -> TimeSeries
select direction (TS triples) =
  TS $ head triples :  filter ((_, _, d) →d == direction) (tail triples)

deltas ::  TimeSeries -> [Float]
deltas (TS triples) = zipWith ((_, v1, _) (_, v2, _) →v2 - v1) triples $ tail triples
```