Varianta A – Examen PP (2020)

1. `(λx y z.corp a b c)` este forma prescurtată de a scrie `(((λx.λy.λz.corp a) b) c)`.

   Fie următoarele definiții în Calcul Lambda:

   `true = λx y.x`

   `false = λx y.y`

   și fie operatorul logic:

   `op = λx y.(x y true)`

   Descrieți pas cu pas (nu efectuați mai multe β-reduceri deodată) comportamentul lui `op` pe valori de tip `true` sau/și `false`, astfel încât să puteți conchide la final că `op` se comportă ca unul dintre operatorii logici cunoscuți (și precizați care este operatorul a cărui identitate a fost ascunsă prin folosirea numelui "`op`").

   Soluție:

   `(op true y) = (λx y.(x y true) true y)` $\rightarrow$ `(true y true) = (λx y.x y true)` $\rightarrow$ `y`

   `(op false y) = (λx y.(x y true) false y)` $\rightarrow$ `(false y true) = (λx y.y y true)` $\rightarrow$ `true`

   Concluzie: `op` este $\Rightarrow$

2. Ce tip de recursivitate are implementarea de mai jos?

```
(define (z x y)
    (if (or (null? x) (null? y))
        '()
        (cons (car x) (cons (car y) (z (cdr y) (cdr x))))))
```

   Definiți (tot în Racket) cât mai eficient funcția `zz` care are același efect cu `z`, dar folosește un alt tip de recursivitate.

   Justificați pe scurt eficiența soluției voastre.

   Soluție:

   Recursivitate pe stivă – o vom transforma în recursivitate pe coadă.

```
(define (zz x y)
    (let z ((x x) (y y) (r '()))
        (if (or (null? x) (null? y))
            (reverse r)
            (z (cdr y) (cdr x) (cons (car y) (cons (car x) r))))))
```

   Eficiență:

   - recursivitatea pe coadă elimină nevoia de a ocupa spațiu pe stiva de apeluri recursive;
   - adăugarea cu `cons` în acumulatorul `r` (și inversarea lui `r` la final) duce la o construcție în timp liniar, spre deosebire de timp pătratic dacă făceam `append` la finalul lui `r`.

3. **Atenție:** Acest exercițiu nu se punctează decât dacă este rezolvat fără a folosi recursivitate explicită (deci nici `letrec` sau `named let`). Soluțiile explicit recursive se punctează cu 0.

   Se dă o listă `L` care conține numere și/sau liste de numere (maxim un nivel de imbricare). Să se definească, în Racket, funcția `two-sums` care întoarce o pereche de valori, reprezentând suma numerelor la "adâncime 0" din `L`, respectiv suma numerelor la "adâncime 1" din `L`.

   ex: `(two-sums '(1 2 (3 4) 5 (6) 7))` va întoarce `'(15 . 13)`, pentru că $1+2+5+7=15$, iar $3+4+6=13$

   Soluție:

```
(define (two-sums L)
    (let ((outside (filter (compose not list?) L))
          (inside (apply append (filter list? L))))
        (cons (apply + outside) (apply + inside))))
```

4. Cum decurge, pas cu pas, evaluarea expresiei:

```
head $ ([1] ++ [2]) ++ [3]
```

presupunând cunoscută implementarea operatorului de concatenare:

```
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Soluție:

```
head $ ([1] ++ [2]) ++ [3]       -- (1)
head $ (1 : ([] ++ [2])) ++ [3] -- (2)
head $ 1 : (([] ++ [2]) ++ [3]) -- (3)
1                                -- (4)
```

5. Sintetizați în Haskell, pas cu pas, tipul funcției `g`:

   ```
   g fs xs = zipWith (\f x -> f x) fs xs
   ```

   cunoscând tipul funcției `zipWith`:

   ```
   zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
   ```

   Soluție:

   ```
   f :: a = d -> e
   x :: b = d
   c = e
   fs :: [a] = [d -> e]
   xs :: [b] = [d]
   g :: [d -> e] -> [d] -> [e]
   ```

6. Supraîncărcați în Haskell operatorul de egalitate pentru funcții unare cu parametru numeric, astfel încât două funcții să fie considerate egale dacă valorile lor coincid în cel puțin 10 puncte din intervalul $1, \ldots, 100$.

   Soluție:

   ```
   instance (Num a, Enum a, Eq b) => Eq (a -> b) where
       f == g = length (filter id [f x == g x | x <- [1..100]]) >= 10
   ```

7. Folosiți pentru scriere `NOT()`, `AND` și `OR` ca operatori logici și `ORICARE` și `EXISTĂ` drept cuantificatori. Scrieți cu caps ca să nu existe confuzii cu limbajul obișnuit. Puteți prescurta numele predicatelor.

   Știind că "Cine împarte (din ceva), parte își face (din acel ceva).", și că

   `NOT(își_face_parte_din(marcel, tort))`, dorim să demonstrăm că `NOT(împarte(marcel, tort))`.

   a) Traduceți proverbul în LPOI (FOL) utilizând numai predicatele menționate în problemă.

   b) Demonstrați **prin metoda rezoluției** concluzia cerută.

   Nu se punctează demonstrațiile care nu folosesc metoda rezoluției.

   Soluție:

   a) $\forall$`X.`$\forall$`Y.împarte(X, Y)` $\Rightarrow$ `își_face_parte(X, Y)`

   b) Clauzele:

   (1) $\neg$`împarte(X, Y)` $\vee$ `își_face_parte_din(X, Y)`

   (2) `împarte(marcel, tort)`

   (3) $\neg$`își_face_parte_din(marcel, tort)`

   (1) + (2) $\{X \leftarrow$ `marcel`$\} \rightarrow$ `își_face_parte_din(marcel, tort)` (4)

   (3) + (4) $\rightarrow \square$ (clauza vidă)

8. Implementați predicatul `filtersorted(+LLIn, -LLOut)`, care primește în `LLIn` o listă de liste de numere și pune în `LLOut` doar acele liste de numere care sunt sortate crescător. De exemplu, `filtersorted([[2, 1], [1, 3], [1, 2, 3, 4], [1, 2, 4, 2]], X)` leagă `X` la `[[1, 3], [1, 2, 3, 4]]`. Explicați cum funcționează implementarea.

   Soluție:

```
filtersorted([], []).
filtersorted([E|LLIn], [E|LLOut]) :- sort(E, E), !, filtersorted(LLIn, LLOut).
filtersorted([_|LLIn], LLOut) :- filtersorted(LLIn, LLOut).
```

9. Folosiți unul sau mai multe dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `med(+L, -Med)`, care găsește elementul median al listei, cu proprietatea că numărul de elemente mai mari decât `Med` este diferit cu cel mult 1 de numărul de elemente mai mici decât `Med`). Nu utilizați recursivitate explicită. Explicați cum funcționează soluția.

   Soluție:

```
med(L, Med) :- member(Med, L), findall(X, (member(X, L), X < Med), L1),
               findall(X, (member(X, L), X > Med), L2), length(L1, LL1),
               length(L2, LL2), abs(LL1-LL2) =< 1.
```

10. În acest exercițiu urmărim crearea unui index de cuvinte dintr-un text: pentru fiecare cuvânt, vom avea o listă cu liniile din text pe care acest apare. În acest scop definim următoarele alias-uri de tip.

    Utilizați următorul schelet:

```
import Data.List
import Data.Char
import Data.Maybe

type Text = String
-- pereche între un cuvânt și o linie din text
type Pair = (String, Int)
-- pereche între un cuvânt și lista tuturor liniilor pe care acesta apare
type Index = [(String, [Int])]
```

    - (10p) Un text constă într-un string de cuvinte care pot fi separate prin spații sau enter (caracterul $\backslash n$). De asemenea, în text pot apărea semne de punctuație precum virgulă, punct, semnul întrebării sau semnul exclamării (după care apare minim un spațiu sau enter).

      Să se implementeze funcția `textToLines` care primește un text și întoarce o listă de perechi de forma: `(linie, text_aflat_pe acea_linie_procesat)`.

      Un text procesat reprezintă textul din care s-au eliminat semnele de punctuație iar literele mari au fost transformate în litere mici.

```
{-
Utile:
-- împarte un text în linii, parsând după '\n'
lines :: String -> [String]
-- transformă o literă mare în literă mică
toLower :: Char -> Char

ex:
*Main> textToLines "A venit,\na venit\ntoamna!"
[(1,"a venit"),(2,"a venit"),(3,"toamna")]
-}

textToLines :: Text -> [(Int, Text)]
textToLines = undefined
```

    - (10p) Să se implementeze funcția `insPair` care inserează o pereche `(cuvânt, linie)` într-un index, fără a crea linii duplicate.

```haskell
{-
ex:
*Main> insPair ("toamna",2) [("a", [1,2])]
[("toamna",[2]),("a",[1,2])]
*Main> insPair ("toamna",2) [("a", [1,2]), ("toamna", [1])]
[("toamna",[2,1]),("a",[1,2])]
*Main> insPair ("toamna",2) [("a", [1,2]), ("toamna", [1,2])]
[("a",[1,2]),("toamna",[1,2])] -- nu s-a mai adăugat încă o dată linia
↪ 2
-}

insPair :: Pair -> Index -> Index
insPair = undefined
```

- (10p) Să se implementeze funcțiile `allPairs` și `textToIndex`. `allPairs` primește un text și creează toate perechile `(cuvânt, linie)` pentru acel text. `textToIndex` primește un text și creează indexul (sortat alfabetic după cuvânt) pentru acel text.

```haskell
{-
Utile: words :: String -> [String]        -- împarte un text în
↪ cuvinte, parsând după spații
Obs: tuplurile sunt ordonabile în Haskell (după primul element din
↪ tuplu, apoi al doilea, etc).

ex:
*Main> allPairs "A venit,\na venit\ntoamna!"
[("a",1),("venit",1),("a",2),("venit",2),("toamna",3)]
*Main> textToIndex "A venit,\na venit\ntoamna!"
[("a",[1,2]),("toamna",[3]),("venit",[1,2])]
-}
allPairs :: Text -> [Pair]
allPairs = undefined

textToIndex :: Text -> Index
textToIndex = undefined
```

Soluție:

```haskell
-1.
textToLines :: Text -> [(Int, Text)]
textToLines = zip [1 ..] . lines . map toLower . filter (`notElem` ",.?!")

--2.
insPair :: Pair -> Index -> Index
insPair (word, line) index = case lookup word index of
  Nothing -> (word, [line]) : index
  Just lines -> if elem line lines then index
                                   else (word, line : lines) : filter ((/=
                                   ↪ word) . fst) index

--3.
allPairs :: Text -> [Pair]
allPairs txt = [ (word, line) | (line, text) <- textToLines txt, word <-
↪ words text ]

textToIndex :: Text -> Index
textToIndex = sort . foldr insPair [] . allPairs
```