B

Calcul Lambda

1. a) Scrieți o λ-expresie care conține fix 2 variabile (x și y), cu 2 apariții fiecare, astfel încât 3 dintre apariții să fie legate și una singură liberă.
   b) Încercuiți aparițiile legate.

   Soluție: `λ x . (λ y . x y)`
   Barem: 2p - λ-expresie validă cu 2 variabile cu 2 apariții fiecare
   6p (condiționate de punctajul anterior) - 3 apariții legate și una liberă
   2p - încercuire corectă

Racket

2. Implementați recursiv funcția find-cycle care adună în mod repetat pătratele cifrelor unui număr până când întâlnește un număr pe care l-a mai prelucrat anterior.
   - ex: `32 -> 9+4 = 13 -> 1+9 = 10 -> 1+0 = 1 -> 1 = 1` (rezultatul apelului este 1)

   Soluție:
```
(define (find-cycle n)
  (let loop ((n n) (acc 0) (prev (list n)))
    (if (zero? n)
        (if (member acc prev)
            acc
            (loop acc 0 (cons acc prev)))
        (loop (quotient n 10)
              (+ acc (sqr (modulo n 10)))
              prev))))
```
   Barem: 2p - extragere cifre
   2p - adunare pătrat cifre
   1p - condiție caz de bază 1 (numărul current nu mai are cifre de adunat)
   4p - repetare de câte ori este necesar
   1p - condiție caz de bază 2 (am ajuns la un număr prelucrat anterior)

3. a) Implementați funcția f care numără de câte ori este adevărat un predicat p între 2 elemente consecutive ale unei liste date L.
   - ex: `p = equal?, L = '(1 1 2 2 2) -> 3 grupuri: '(1 1), '(2 2), '(2 2)`
   - **Constrângeri:** - f trebuie să fie curry într-un mod util punctului b)
   - folosiți funcționale, nu folosiți recursivitate explicită
   b) Implementați funcția f1 care determină câte elemente ale listei argument sunt pătratul succesorului lor.
   - **Constrângeri:** - f1 trebuie să fie o aplicație parțială a lui f
   - folosiți funcții anonime

   Soluție:
```
(define (f p)
  (λ (L)
    (foldl (λ (x1 x2 acc) (if (p x1 x2) (+ 1 acc) acc))
           0
           (drop-right L 1)
           (cdr L))))
```

```
(define f1 (f (λ (x y) (= x (sqr y)))))
```

Barem: 2p - forma curry (întâi p și apoi L)
1p - apelarea predicatului pe x1, x2
2p - numărarea corectă a perechilor care satisfac p
2p - utilizare corectă funcționale (ca număr și tip al argumentelor, cât timp acestea au sens)
1p - f1 este o aplicație parțială a lui f
2p - funcție anonimă corectă

4. Implementați funcția s care primește o listă L_0 și întoarce fluxul L_0, L_1, L_2, ... unde L_i se obține din L_{i-1} prin decrementarea elementelor de pe poziții impare. Considerați lista indexată de la 0.
- ex: L_0 = `'(1 1 1 1 1)` -> `'(1 1 1 1 1), '(1 0 1 0 1), '(1 -1 1 -1 1) …`

Soluție:
```
(define (s L0)
  (define indices (range (length L0)))
  (define (process x pos)
    (if (odd? pos) (sub1 x) x))

  (let loop ((L L0))
    (stream-cons
     L
     (loop (map process L indices)))))
```
Barem: 2p - folosire corectă a interfeței pentru fluxuri (stream-cons, funcționale pe fluxuri, etc.)
3p - verificare paritate și incrementare poziție
2p - obținerea listei următoare din lista curentă
3p - construcție flux final (explicit sau implicit)

Haskell

5. Scrieți o funcție care are signatura (Ord b, Num a) => (a -> b) -> [a] -> [(a, b)].

Soluție:
```
f fun lst = [ (x, fun x) | x <- lst, fun x < fun (x * x) ]
```
Barem: 1p - primul parametru de tip (a -> b) (*)
1p - al doilea parametru de tip [a] (**)
2p - rezultat de tip [(a, b)]
3p (condiționate de (*)) - Ord b
3p (condiționate de (**)) - Num a

6. Modelăm un joc de ghicire a unui număr target, în care fiecare tentativă primește un feedback de tip "prea mare", "prea mic", sau "la fix", folosind următoarele tipuri de date:

```
data GuessingGame = GG { target :: Int, guesses :: [Guess] } deriving Show
data Result = TooLow | TooHigh | SpotOn deriving Show
data Guess = Guess { attempt :: Int, result :: Result } deriving Show
data OrdGuess = OrdGuess { guess :: Guess, reference :: Int } deriving
                                                              Show
```

Tipul OrdGuess reține atât o încercare guess cât și target-ul (referința) cu care a fost comparată această încercare.
Adăugați tipul OrdGuess la clasa Ord astfel încât valorile tipului să fie ordonate după cât de apropiate sunt încercările de target (mai aproape = mai mic). Asigurați-vă că îndepliniți toate precondițiile adăugării tipului la clasa Ord.

Soluție:

```
instance Eq OrdGuess where
  OrdGuess (Guess g _) t == OrdGuess (Guess g' _) t' = abs (g-t) == abs (g'-t')
instance Ord OrdGuess where
  OrdGuess (Guess g _) t <= OrdGuess (Guess g' _) t' = abs (g-t) <= abs (g'-t')
```

Barem: 3p - instanțierea clasei Eq (a.î. două valori sunt egale dacă sunt la fel de apropiate de target)
1p - sintaxă corectă instanțiere
1p - alegere de a implementa <= sau compare
1p - abs (sau ceva echivalent)
2p - extragerea g și t prin pattern matching sau funcții selector
2p - compararea diferențelor

Logică

7. Aduceți propoziția ¬(P ∧ Q) ⟹ (¬P ∨ ¬Q) la forma clauzală, precizând pașii efectuați.

Soluție (și barem):

(eliminare implicații): ¬(¬(P∧Q)) ∨(¬P ∨¬Q) - 3p
(introducere negații): (P ∧Q) ∨(¬P ∨¬Q) - 3p
(distribuire ∨ prin ∧): (P ∨¬P ∨¬Q) ∧(Q ∨¬P ∨¬Q) - 3p
(transformare în clauze): {P, ¬P, ¬Q}, {Q, ¬P, ¬Q} (precizarea pașilor valorează 1p)

Prolog

8. Implementați predicatul highest_increasing(+L, -S) care determină cea mai mare sumă de elemente consecutive strict crescătoare din lista L. Util: funcția max/2 (ex utilizare: M is max(2, 3)).
- ex: `highest_increasing([1,2,3,3,4,3,2,2,2,3,3,3,3], S). -> S = 7`

Soluție:

```
highest_increasing([], 0).
highest_increasing([X|Xs], S) :- helper(Xs, X, X, 0, S).
% helper(+L, +PrevElem, +CurrentSum, +MaxSum, -Result)
helper([], _, Crt, Max, S) :- S is max(Crt, Max).
helper([X|Xs], Prev, Crt, Max, S) :- Prev < X, C is Crt + X, helper(Xs, X, C,
Max, S), !.
helper([X|Xs], _, Crt, Max, S) :- M is max(Crt, Max), helper(Xs, X, X, M, S), !.
```

Barem: 1p - listă vidă
2p - caz de bază listă nevidă
3p - caz în care suma continuă (1p identificare caz, 1p adunare, 1p apel recursiv)
3p - caz în care suma se resetează (1p identificare, 1p resetări, 1p apel recursiv)
1p - condiții mutual exclusive

9. Fie un program care gestionează o bibliotecă, modelată prin fapte de tip carte(Titlu, Autor, An, Gen).
- ex: `carte('Dune', 'Frank Herbert', 1965, science_fiction).`
Implementați predicatul genre_titles(?Genre, -Titles), care determină lista titlurilor de un anumit gen. Dacă parametrul Genre este o variabilă, atunci predicatul se satisface în mod repetat pentru fiecare gen din bibliotecă (legând Genre la un gen și Titles la lista titlurilor de acest gen).
- **Constrângeri:** - folosiți metapredicate, nu folosiți recursivitate explicită

Soluție:

```
genre_titles(Genre, Titles) :-
```

```
bagof(Title, A^Y^book(Title, A, Y, Genre), Titles).
```

Barem: 2p - soluție (probabil bagof) pentru ca fiecare gen să fie selectat o singură dată
1p - argumente de tip template, goal, bag
2p - template corect (titlu)
2p - goal corect (fapt book cu titlul/genul marcat cu aceleași variabile ca în template/antet)
1p - bag = Titles
2p - A^Y^

Problema (Haskell)

10. Pentru jocul și tipurile descrise la exercițiul 6:

a) (10p) Implementați funcția isWin :: GuessingGame -> (Bool, Maybe Int), care primește un joc și întoarce o pereche între:
- un boolean (adevărat dacă jocul a fost câștigat, fals altfel)
- numărul de încercări efectuate până la victorie - în caz că jocul a fost câștigat
(se folosește un Maybe Int pentru a acoperi și cazul în care jocul nu a fost câștigat).

- **Observație:** un joc se încheie odată cu ghicirea numărului, o încercare reușită nu va fi urmată în listă de alte încercări.

b) (5p) Implementați funcția checkGuess :: Guess -> Maybe Int -> Maybe Int -> (Bool, Maybe Int, Maybe Int), care primește o încercare și două limite (inferioară, respectiv superioară) între care ar trebui să se afle încercarea conform istoricului de încercări, și întoarce un triplet între:
- un boolean - adevărat dacă și numai dacă încercarea curentă a fost între limitele date
- noua limită inferioară (actualizată conform feedback-ului primit pentru încercarea curentă)
- noua limită superioară (actualizată conform feedback-ului primit pentru încercarea curentă)

- **Observație:** Limitele sunt modelate cu un Maybe pentru că este nevoie de minim o încercare prea mică/mare pentru a avea o limită inferioară/superioară. Trebuie tratat și cazul în care limitele (una sau ambele) nu au fost încă setate.

(5p) Apoi implementați funcția isPlayerLogical :: GuessingGame -> Bool, care verifică dacă șirul de încercări este logic (nu s-a făcut niciodată o încercare inutilă - în afara limitelor stabilite prin încercările anterioare).

c) (10p) Implementați funcția bestGuess :: GuessingGame -> Maybe Guess care determină cea mai bună încercare efectuată de-a lungul jocului, adică încercarea cea mai apropiată de target (la egalitate, se poate întoarce oricare dintre opțiuni). Folosim un Maybe pentru a acoperi și situația dinaintea efectuării primei încercări.

Soluție:

```
isWin :: GuessingGame -> (Bool, Maybe Int)
isWin (GG _ guesses)
  | Guess _ SpotOn <- last guesses = (True, Just $ length guesses)
  | otherwise = (False, Nothing)

checkGuess :: Guess -> Maybe Int -> Maybe Int -> (Bool, Maybe Int, Maybe Int)
checkGuess (Guess g TooLow) (Just inf) sup = (g > inf, Just (max g inf), sup)
checkGuess (Guess g TooLow) Nothing sup = (True, Just g, sup)
checkGuess (Guess g TooHigh) inf (Just sup) = (g < sup, inf, Just (min g sup))
checkGuess (Guess g TooHigh) inf Nothing = (True, inf, Just g)
checkGuess (Guess g SpotOn) inf sup = (True, Just (g-1), Just (g+1))

isPlayerLogical :: GuessingGame -> Bool
isPlayerLogical game = process (guesses game) Nothing Nothing
  where
  process [] _ _ = True
  process (g:gs) inf sup =
```

```
    let (ok, inf', sup') = checkGuess g inf sup
    in  ok && process gs inf' sup'

bestGuess :: GuessingGame -> Maybe Guess
bestGuess (GG _ []) = Nothing
bestGuess (GG target guesses) = Just g
  where OrdGuess g _ = minimum [ OrdGuess guess target | guess <- guesses ]
```

Barem: a) 6p - caz câștig (3p identificare caz, 1p calcul număr de încercări, 2p retur)
2p - caz non-câștig
2p - utilizare corectă Maybe
b) checkGuess: 2p - cazul în care limita de interes există (1p pattern matching, 1p retur)
2p - cazul … nu există (1p pattern matching, 1p retur)
1p - repetarea procedeului pentru cele 2 tipuri de Result (TooLow, TooHigh)
0p - cazul SpotOn este opțional
isPlayerLogical: 1p - caz de bază (toate încercările au fost logice)
2p - fals dacă se găsește o încercare ilogică
2p - parcurgerea și testarea listei de încercări
c) 2p - cazul în care nu există încercări
6p - parcurgerea listei de încercări (3p) și reținerea celei mai apropiate de target (3p)
(obs: soluția oficială folosește instanțierea de la exercițiul 6, dar se poate și fără)
2p - utilizare corectă Maybe