

## ALTELE RACKET

```
(define (greater-sum L)
  (filter (λ (S) ; extrag din lista L acele liste pentru care
    (≥=
      (foldr + 0 S) ; suma e mai mare
      (foldr * 1 S) ; decat produsul
    )))
  L))
```

```
(define (greater-sum L)
  (let ((check? (λ (l)
    (≥= (foldr + 0 l) (foldr * 1 l)
  ))))
  (filter (λ (p) (check? p)) L)
  ))
```

```
(define (count-occ L1 L2)
  (let ([count (λ (x)
    (length (filter (λ (p) (equal? x p)) L2)))
  ])) ; functie ce determina numarul de aparitii ale unui numar in lista L2
  (map (λ (y) (cons y (count y))) L1) ; la final construiesc rezultatul adaugand la fiecare
  ))
```

```
(define (zip L1 L2)
  (map cons L1 L2))
```

```
(define (unzip L)
  (cons
    (foldr (λ(x acc) (cons (car x) acc)) '() L)
    (list (foldr (λ(x acc) (cons (cdr x) acc)) '() L))))
```

```
(define (divisors L1 L2)
  (let ((find-divs (λ(x) ;consider o functie anonyma
```

```
;ce imi extrage toate elementele din L2 ce sunt divizori  
;ai lui "x".  
(filter (λ(num) (= (modulo x num) 0)) L2)  
)))  
(map (λ(y) (cons y (list (find-divs y)))) L1) ;adaug la fiecare element  
;corespunzator din L1 lista formata cu divizorii sai din L2.  
))
```

## CLASE IN HASKELL

```

length_ [] = 0
length_ (_:xs) = 1 + length_ xs
length :: [a] -> Int
map :: (a -> b) -> [a] -> [b]

elem _ []      = False
elem x (y:ys) = x == y || elem x ys
elem :: Eq a => a -> [a] -> Bool

Clase
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

data Person = Person {name :: String, cnp :: Integer}
instance Eq Person where
  Person name1 cnp1 == Person name2 cnp2 = name1 == name2 && cnp1
  == cnp2
  p1 /= p2 = not (p1 == p2)
data BST a = Empty | Node a (BST a) (BST a)
instance Eq a => Eq (BST a) where
  Empty == Empty = True
  Node info1 l1 r1 == Node info2 l2 r2 = info1 == info2 && l1 ==
  l2 && r1 == r2
  == = False
  t1 /= t2 = not (t1 == t2)

class (Eq a) => Ord a where
  compare           :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min          :: a -> a -> a

  compare x y = if x == y then EQ
                 else if x <= y then LT
                 else GT

  x < y = case compare x y of { LT -> True; _ -> False }
  x <= y = case compare x y of { GT -> False; _ -> True }
  x > y = case compare x y of { GT -> True; _ -> False }
  x >= y = case compare x y of { LT -> False; _ -> True }

  max x y = if x <= y then y else x
  min x y = if x <= y then x else y

class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  a /= b = not (a == b)

```

```

instance Eq Person where
    Person name1 cnp1 == Person name2 cnp2 =
        name1 == name2 && cnp1 == cnp2

```

```

class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
    x == y = not (x /= y)

```

### Clase predefinite

Ord - pentru tipuri care pot fi ordonate - definește funcții precum <, >, <=, etc.

Show - pentru tipuri care pot fi reprezentate ca String-uri - principala funcție este show. Această funcție este folosită și de consola GHCi atunci când afișează rezultatele.

### Deriving

```
data BST a = Empty | Node a (BST a) (BST a) deriving (Eq)
```

```
treeMap :: (a -> b) -> BST a -> BST b
```

```
treeMap f Empty = Empty
```

```
treeMap f (BST info l r) = BST (f info) (treeMap f l) (treeMap f r)
```

```
data Maybe a = Nothing | Just a
```

```
maybeMap :: (a -> b) -> Maybe a -> Maybe b
```

```
maybeMap f Nothing = Nothing
```

```
maybeMap f (Just x) = Just (f x)
```

```
class Functor container where
```

```
    fmap :: (a -> b) -> container a -> container b
```

```
instance Functor BST where
```

```
    fmap f Empty = Empty
```

```
    fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)
```

{-

Clasa Container reprezintă o clasă folosită pentru enumerarea elementelor unei structuri de date (listă, arbore, graf, etc.)

-}

```
class Container t where
```

```
    contents :: t a -> [a]
```

{-

Clasa Invertible reprezintă o clasă folosită pentru inversarea ordinii de apariție a elementelor unei structuri de date (listă, arbore, etc.)

-}

```
class Invertible a where
```

```
    invert :: a -> a
```

```

{- BST (Binary Search Tree) reprezintă un tip de date ce modelează
   un arbore binar de căutare. Ați definit funcționalitățile
   acestui
   tip de date în cadrul laboratorului 8.

   Funcționalitățile sunt deja definite în cadrul acestui
   laborator.
-}

data BST a = BSTNod
  { vl :: a
  , lt :: BST a
  , rt :: BST a
  } | BSTNil

insertElem :: (Ord a, Eq a) => BST a -> a -> BST a
insertElem BSTNil elem = BSTNod elem BSTNil BSTNil
insertElem root@(BSTNod value left right) elem
  | value == elem = root
  | value < elem = BSTNod value left (insertElem right elem)
  | value > elem = BSTNod value (insertElem left elem) right

findElem :: (Ord a, Eq a) => BST a -> a -> Maybe a
findElem BSTNil _ = Nothing
findElem (BSTNod value left right) elem
  | value == elem = Just value
  | value < elem = findElem right elem
  | value > elem = findElem left elem

size :: BST a -> Int
size BSTNil = 0
size (BSTNod _ left right) = 1 + size left + size right

height :: BST a -> Int
height BSTNil = 0
height (BSTNod elem left right) = 1 + max (height left) (height
right)

inorder :: BST a -> [a]
inorder BSTNil = []
inorder (BSTNod elem left right) = inorder left ++ [elem] ++ inorder
right

{- Arbore folosit pentru testare
-}

root = foldl insertElem BSTNil [7, 4, 12, 2, 3, 1, 10, 15, 8]

{- 1. Instantiați Eq pentru tipul de date BST, prin care se

```

```
    verifică dacă doi arbori de acoperire sunt identici.  
- }  
  
instance Eq a => Eq (BST a) where  
    (==) BSTNil BSTNil = True  
    (==) (BSTNod e1 l1 r1) (BSTNod e2 l2 r2) = e1 == e2 && l1 == l2  
&& r1 == r2  
    (==) _ _ = False
```

```
{ -  
2. Instantiați Show pentru tipul de date BST.  
Fiecare nivel de adâncime în arbore va fi reprezentat de un număr  
corespunzător de tab-uri. De exemplu pentru nivelul 2 de adâncime  
se vor adăuga două tab-uri.
```

```
Fiecare element din arbore va avea linia sa, adică câte un  
element  
din arbore pe o linie.
```

```
- }
```

```
printLevel :: Show a => Char -> Int -> BST a -> [Char]  
printLevel _ BSTNil = ""  
printLevel tab level (BSTNod root left right) = replicate level tab  
++ show root ++ "\n"  
++ printLevel tab (level + 1) left  
++ printLevel tab (level + 1) right  
  
instance Show a => Show (BST a) where  
    show BSTNil = ""  
    show node@BSTNod{} = printLevel '\t' 0 node
```

```
{ -  
3. Instantiați Ord pentru tipul de date BST.  
Criteriul de comparare a doi arbori va fi după înălțimea lor  
(funcția height).
```

```
Trebuie implementată funcția (<=)  
- }
```

```
instance Ord a => Ord (BST a) where  
    (<=) t1 t2 = height t1 <= height t2
```

```
{ -  
4. Instantiați Invertible pentru tipul de date BST.  
Funcția invert, în acest caz, va inversa ordinea subarborilor.  
- }
```

```
instance Invertible (BST a) where  
    invert BSTNil = BSTNil
```

```
    invert (BSTNod a left right) = BSTNod a (invert right) (invert left)
```

```
{ -
```

5. Instantiați Functor pentru tipul de date BST.

Funcția fmap este similară funcției map, prin care se aplică o funcție f tuturor elementelor din structură.

```
- }
```

```
instance Functor BST where
```

```
    fmap f BSTNil = BSTNil
```

```
    fmap f (BSTNod a left right) = BSTNod (f a) (fmap f left) (fmap f right)
```

```
{ -
```

6. Instantiați Foldable pentru tipul de date BST.

Funcția foldr are aceeași funcționalitate atunci când ea este aplicată pe liste.

Nu trebuie să implementați și foldl, clasa Foldable nu acoperă și această funcție.

```
- }
```

```
instance Foldable BST where
```

```
    foldr f acc BSTNil = acc
```

```
    foldr f acc (BSTNod value left right) = foldr f (f value newAcc) left
```

```
        where newAcc = foldr f acc right
```

```
{ -
```

7. Instantiați Container pentru tipul de date BST.

Pentru implementarea funcției contents o să folosiți funcția foldr, implementată la exercițiul anterior.

```
- }
```

```
instance Container BST where
```

```
    contents tree = foldr (:) [] tree
```

```
{ -
```

8. Implementați sizeFold, care calculează numărul de elemente din cadrul unui arbore binar de căutare (există deja funcția size, care face același lucru, definită mai sus).

```
- }
```

```
sizeFold :: BST a -> Int
```

```
sizeFold tree = foldr (\_ acc -> acc + 1) 0 tree
```



## Close, tipuri în spirit Haskell

class Eq a where

(==) :: a → a → Bool

instance Eq Integer where

$$x == y = \text{integerEq } x y.$$

instance (Eq a) ⇒ Eq (Tree a) where

leaf a == leaf b = a == b

(Branch l1 r1) == (Branch l2 r2) = (l1 == l2) ∧ (r1 == r2)

— == — = False

class (Eq a) ⇒ Ord a where

(<), (<=), (>=), (>) :: a → a → Bool

max, min :: a → a

$$x < y = x <= y \wedge x \neq y.$$

instance Num a where ...

data Point = Pt { pointX, pointY :: Float }.

pointX :: Point → Float -- data do

-- la fel și la pointY

absPoint :: Point → Float -- functie.

absPoint (Pt { pointX = x, pointY = y }) = sqrt (x^2 + y^2)

Close standard Haskell

{ data Ordering = EQ | LT | GT

show :: (Show a) ⇒ a → String

instance Show a ⇒ Show (Tree a) where

ShowFor x = showTree x. →

instance Show a => Show(Tree a) where

Show t = showTree

ShowTree :: (Show a) => Tree a -> Show

shows (Leaf x) = shows x

showsTree (Branch l r) = l : ) . showsTree l . ('!' : ) . showsTree r . ('!' : )

data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Eq

instance Ord a => Ord (Tree a) where

(Leaf -) <= (Branch -) = True

(Leaf x) <= (Leaf y) = x <= y

(Branch -) <= (Leaf -) = False

(Branch l r) <= (Branch l' r') = l == l' && r <= r' || r <= l'

1. Reduceti expresia lambda  $(\lambda y. \lambda x. (y \ y) \ (x \ x))$

Solutie:

$$\xrightarrow{\alpha} (\lambda y. \lambda z. (y \ y) \ (x \ x)) \xrightarrow{\lambda y.} \lambda z. ((x \ x) \ (x \ x))$$

1. Reduceti expresia lambda  $(\lambda x. (\lambda x. (x \ x) \ \lambda x. x) \ (x \ x))$

Solutie:

$$\xrightarrow{\text{primul } \lambda x.} (\lambda x. (x \ x) \ \lambda x. x) \rightarrow (\lambda x. x \ \lambda x. x) \rightarrow \lambda x. x$$

1.  $(\lambda x \ y \ z. \text{corp a b c})$  este forma prescurtată de a scrie  $((\lambda x. \lambda y. \lambda z. \text{corp a}) \ b) \ c$ .

Fie următoarele definiții în Calcul Lambda:

$$\text{true} = \lambda x \ y. x$$

$$\text{false} = \lambda x \ y. y$$

și fie operatorul logic:

$$\text{op} = \lambda x \ y. (x \ y \ \text{true})$$

Descrieți pas cu pas (nu efectuați mai multe  $\beta$ -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").

Solutie:

$$(\text{op true } y) = (\lambda x \ y. (x \ y \ \text{true}) \ \text{true } y) \rightarrow (\text{true } y \ \text{true}) = (\lambda x \ y. x \ y \ \text{true}) \rightarrow y$$

$$(\text{op false } y) = (\lambda x \ y. (x \ y \ \text{true}) \ \text{false } y) \rightarrow (\text{false } y \ \text{true}) = (\lambda x \ y. y \ y \ \text{true}) \rightarrow \text{true}$$

Concluzie: op este  $\Rightarrow$

1.  $(\lambda x \ y \ z. \text{corp a b c})$  este forma prescurtată de a scrie  $((\lambda x. \lambda y. \lambda z. \text{corp a}) \ b) \ c$ .

Fie următoarele definiții în Calcul Lambda:

$$\text{true} = \lambda x \ y. x$$

$$\text{false} = \lambda x \ y. y$$

și fie operatorul logic:

$$\text{op} = \lambda x \ y. (x \ \text{true } y)$$

Descrieți pas cu pas (nu efectuați mai multe  $\beta$ -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").

Solutie:

$$(\text{op true } y) = (\lambda x \ y. (x \ \text{true } y) \ \text{true } y) \rightarrow (\text{true true } y) = (\lambda x \ y. x \ \text{true } y) \rightarrow \text{true}$$

$$(\text{op false } y) = (\lambda x \ y. (x \ \text{true } y) \ \text{false } y) \rightarrow (\text{false true } y) = (\lambda x \ y. y \ \text{true } y) \rightarrow y$$

Concluzie: op este or

1.  $(\lambda x y z. \text{corp} a b c)$  este forma prescurtată de a scrie  $((\lambda x. \lambda y. \lambda z. \text{corp} a) b) c$ .

Fie următoarele definiții în Calcul Lambda:

`null =  $\lambda x. \text{true}$`

`cons =  $\lambda x y z. (z \ x \ y)$`

și fie operatorul pe liste:

`op =  $\lambda L. (L \ \lambda x y. \text{false})$`

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

*Soluție:*

$$(\text{op } \text{null}) = (\lambda L. (L \ \lambda x y. \text{false}) \ \lambda x. \text{true}) \rightarrow (\lambda x. \text{true} \ \lambda x y. \text{false}) \rightarrow \text{true}$$

$$(\text{op } (\text{cons } a b)) = (\text{op } (\lambda x y z. (z \ x \ y) \ a \ b)) \rightarrow (\text{op } \lambda z. (z \ a \ b))$$

$$(\text{op } \lambda z. (z \ a \ b)) = (\lambda L. (L \ \lambda x y. \text{false}) \ \lambda z. (z \ a \ b)) \rightarrow (\lambda z. (z \ a \ b) \ \lambda x y. \text{false}) \rightarrow (\lambda x y. \text{false} \ a \ b) \rightarrow \text{false}$$

Concluzie: `op` este `null`?

1.  $(\lambda x y z. \text{corp} a b c)$  este forma prescurtată de a scrie  $((\lambda x. \lambda y. \lambda z. \text{corp} a) b) c$ .

Fie următoarea definiție în Calcul Lambda:

`cons =  $\lambda x y z. (z \ x \ y)$`

și fie operatorul pe liste nevide:

`op =  $\lambda L. (L \ \lambda x y. y)$`

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

*Soluție:*

$$(\text{op } (\text{cons } a b)) = (\text{op } (\lambda x y z. (z \ x \ y) \ a \ b)) \rightarrow (\text{op } \lambda z. (z \ a \ b))$$

$$(\text{op } \lambda z. (z \ a \ b)) = (\lambda L. (L \ \lambda x y. y) \ \lambda z. (z \ a \ b)) \rightarrow (\lambda z. (z \ a \ b) \ \lambda x y. y) \rightarrow (\lambda x y. y \ a \ b) \rightarrow b$$

Concluzie: `op` este `cdr`

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

*Soluție:*

$$\rightarrow (\lambda x. (x \ z) \ \lambda x. x) \rightarrow (\lambda x. x \ z) \rightarrow z$$

1. Reduceti expresia lambda  $E = (y \ (\lambda x. \lambda x. x \ (\lambda y. y \ y)))$

*Solutie:*

$$\rightarrow (y \ (\lambda x. \lambda x. x \ y)) \rightarrow (y \ \lambda x. x)$$

sau

$$\rightarrow (y \ \lambda x. x)$$

1. Reduceti la forma normală următoarea expresie, ilustrând pașii de reducere:

$$\lambda x. \lambda y. ((\lambda x. \lambda y. x \ (y \ x)) \ (x \ y))$$

*Solutie:*

$$\lambda x. \lambda y. ((\lambda x. \lambda y. x \ (y \ x)) \ (x \ y)) \xrightarrow{\alpha} \lambda x. \lambda y. ((\lambda x. \lambda z. x \ (y \ x)) \ (x \ y)) \xrightarrow{\beta}$$

$$\lambda x. \lambda y. (\lambda z. (y \ x) \ (x \ y)) \xrightarrow{\beta} \lambda x. \lambda y. (y \ x)$$

1. Reduceti la forma normală următoarea expresie, ilustrând pașii de reducere:

$$\lambda x. \lambda y. ((\lambda x. \lambda y. y \ (x \ y)) \ (y \ x))$$

*Solutie:*

$$\lambda x. \lambda y. ((\lambda x. \lambda y. y \ (x \ y)) \ (y \ x)) \xrightarrow{\beta} \lambda x. \lambda y. (\lambda y. y \ (y \ x)) \xrightarrow{\beta} \lambda x. \lambda y. (y \ x)$$

1. Determinati forma normală pentru următoarea expresie, ilustrând pașii de reducere:

$$((\lambda x. \lambda y. \lambda z. (x \ y) \ \lambda x. x) \ z)$$

*Solutie:*

$$((\lambda \underline{x}. \lambda y. \lambda z. (\underline{x} \ y) \ \lambda x. x) \ z) \xrightarrow{\beta} (\lambda \underline{y}. \lambda z. (\lambda x. x \ \underline{y}) \ z) \xrightarrow{\alpha} (\lambda \underline{y}. \lambda w. (\lambda x. x \ \underline{y}) \ z) \xrightarrow{\beta} \lambda w. (\lambda \underline{x}. \underline{x} \ z) \xrightarrow{\beta} \lambda w. z$$

1. Determinati forma normală pentru următoarea expresie, ilustrând pașii de reducere:

$$((\lambda x. \lambda y. \lambda z. (y \ x) \ y) \ \lambda z. z)$$

*Solutie:*

$$((\lambda \underline{x}. \lambda y. \lambda z. (y \ \underline{x}) \ y) \ \lambda z. z) \xrightarrow{\alpha} ((\lambda \underline{x}. \lambda w. \lambda z. (w \ \underline{x}) \ y) \ \lambda z. z) \xrightarrow{\beta} (\lambda \underline{w}. \lambda z. (\underline{w} \ y) \ \lambda z. z) \xrightarrow{\beta} \lambda z. (\lambda \underline{z}. \underline{z} \ y) \xrightarrow{\beta} \lambda z. y$$

1. Determinati forma normală pentru următoarea expresie, ilustrând pașii de reducere:

$$((\lambda x. \lambda y. \lambda z. (x \ y) \ \lambda x. y) \ a)$$

*Solutie:*

$$((\lambda \underline{x}. \lambda y. \lambda z. (\underline{x} \ y) \ \lambda x. y) \ a) \xrightarrow{\alpha} ((\lambda \underline{x}. \lambda w. \lambda z. (\underline{x} \ w) \ \lambda x. y) \ a) \xrightarrow{\beta} (\lambda \underline{w}. \lambda z. (\lambda x. y \ \underline{w}) \ a) \xrightarrow{\beta} \lambda z. (\lambda x. y \ a) \xrightarrow{\beta} \lambda z. y$$

1. Determinati forma normală pentru următoarea expresie, ilustrând pașii de reducere:

$$((\lambda z. \lambda y. \lambda x. (y \ z) \ y) \ \lambda y. y)$$

*Solutie:*

$$((\lambda \underline{z}. \lambda y. \lambda x. (y \ \underline{z}) \ y) \ \lambda y. y) \xrightarrow{\alpha} ((\lambda \underline{z}. \lambda w. \lambda x. (w \ \underline{z}) \ y) \ \lambda y. y) \xrightarrow{\beta} (\lambda \underline{w}. \lambda x. (\underline{w} \ y) \ \lambda y. y) \xrightarrow{\beta} \lambda x. (\lambda \underline{y}. \underline{y} \ y) \xrightarrow{\beta} \lambda x. y$$

1. Ilustrați cele două posibile secvențe de reducere pentru expresia:  $(\lambda y.(\lambda x.\lambda y.x\ y)\ 2)$

*Soluție:*

- $(\underline{\lambda y}.(\lambda x.\lambda y.x\ \underline{y})\ 2) \xrightarrow[\beta]{stanga-dreapta} (\underline{\lambda x}.\lambda y.\underline{x}\ 2) \rightarrow_\beta \lambda y.2$
- $(\lambda y.(\lambda x.\underline{\lambda y}.x\ y)\ 2) \rightarrow_\alpha (\lambda y.(\underline{\lambda x}.\lambda z.\underline{x}\ y)\ 2) \xrightarrow[\beta]{dreapta-stanga} (\lambda y.\lambda z.y\ 2) \rightarrow_\beta \lambda z.2$

1. Ilustrați cele două posibile secvențe de reducere pentru expresia:  $(\lambda x.(\lambda y.\lambda x.y\ x)\ 5)$

*Soluție:*

- $(\underline{\lambda x}.(\lambda y.\lambda x.y\ \underline{x})\ 5) \xrightarrow[\beta]{stanga-dreapta} (\underline{\lambda y}.\lambda x.\underline{y}\ 5) \rightarrow_\beta \lambda x.5$
- $(\lambda x.(\lambda y.\underline{\lambda x}.y\ x)\ 5) \rightarrow_\alpha (\lambda x.(\underline{\lambda y}.\lambda z.\underline{y}\ x)\ 5) \xrightarrow[\beta]{dreapta-stanga} (\lambda x.\lambda z.x\ 5) \rightarrow_\beta \lambda x.5$

1. Reduceti la forma normală expresia:

$$(\lambda y.((\lambda x.\lambda y.x\ y)\ \Omega)\ \lambda x.\lambda y.y)$$

*Soluție:*

$$\begin{aligned} & (\underline{\lambda y}.((\lambda x.\lambda y.x\ y)\ \Omega)\ \underline{\lambda x}.\lambda y.y) \\ & \rightarrow ((\underline{\lambda x}.\lambda y.x\ \underline{\lambda x}.\lambda y.y)\ \Omega) \\ & \rightarrow (\underline{\lambda y}.\lambda x.\lambda y.y\ \underline{\Omega}) \\ & \rightarrow \lambda x.\lambda y.y \end{aligned}$$

1. Reduceti la forma normală expresia:

$$(((\lambda x.\lambda y.\lambda z.y\ \lambda x.x)\ (\lambda z.\lambda t.z\ z))\ \Omega)$$

*Soluție:*

$$\begin{aligned} & (((\lambda x.\lambda y.\lambda z.y\ \lambda x.x)\ (\lambda z.\lambda t.z\ z))\ \Omega) \\ & \rightarrow (((\underline{\lambda x}.\lambda y.\lambda z.y\ \underline{\lambda x}.\lambda t.z)\ (\lambda z.\lambda t.z\ z))\ \Omega) \\ & \rightarrow ((\underline{\lambda y}.\lambda z.y\ (\lambda z.\lambda t.z\ z))\ \Omega) \\ & \rightarrow (\underline{\lambda w}.\lambda z.\lambda t.z\ z\ \underline{\Omega}) \\ & \rightarrow (\underline{\lambda v}.\lambda t.\underline{v}\ z) \\ & \rightarrow \lambda t.z \end{aligned}$$

1. Reduceti expresia E la forma normală:  $E \equiv ((\lambda y.(\lambda x.\lambda y.x\ y)\ \lambda x.x)\ \Omega)$

*Soluție:*

$$((\lambda y.(\lambda x.\lambda y.x\ y)\ \lambda x.x)\ \Omega) \rightarrow ((\lambda x.\lambda y.x\ \lambda x.x)\ \Omega) \rightarrow (\lambda y.\lambda x.x\ \Omega) \rightarrow \lambda x.x$$

1. Reduceti expresia E la forma normală:

$$E \equiv ((\lambda x.\lambda y.\lambda x.x\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))\ y)$$

*Soluție:*

$$\lambda x.x$$

## FUNCTII RKT

1. `list-length`: Returnează numărul de elemente dintr-o listă de liste.

```
(define (list-length lst)
  (if (null? lst)
    0
    (+ (length (car lst)) (list-length (cdr lst))))))
```

2. `list-flatten`: Aplatizează o listă de liste într-o singură listă.

```
(define (list-flatten lst)
  (cond
    [(null? lst) '()]
    [(list? (car lst)) (append (list-flatten (car lst)) (list-flatten (cdr lst)))]
    [else (cons (car lst) (list-flatten (cdr lst))))]))
```

3. `list-reverse`: Inversează ordinea elementelor într-o listă de liste.

```
(define (list-reverse lst)
  (cond
    [(null? lst) '()]
    [(list? (car lst)) (append (list-reverse (cdr lst)) (list (list-reverse (car lst))))]
    [else (cons (car lst) (list-reverse (cdr lst))))]))
```

4. `list-append`: Concatenează două liste de liste.

```
(define (list-append lst1 lst2)
  (append lst1 lst2))
```

5. `list-member`: Verifică dacă un element se găsește într-o listă de liste.

```
(define (list-member item lst)
  (cond
    [(null? lst) #f]
    [(list? (car lst)) (or (list-member item (car lst)) (list-member item (cdr lst)))]
    [(equal? item (car lst)) #t]
    [else (list-member item (cdr lst))]))
```

6. `list-filter`: Filtrează elementele dintr-o listă de liste pe baza unui predicat.

```
(define (list-filter pred lst)
```

```
  (cond
```

```
    [(null? lst) '()]
```

```
    [(list? (car lst)) (cons (list-filter pred (car lst)) (list-filter pred (cdr lst)))]
```

```
    [(pred (car lst)) (cons (car lst) (list-filter pred (cdr lst)))]
```

```
    [else (list-filter pred (cdr lst))]))
```

7. `list-map`: Aplică o funcție pe fiecare element dintr-o listă de liste și returnează rezultatele.

```
(define (list-map func lst)
```

```
  (cond
```

```
    [(null? lst) '()]
```

```
    [(list? (car lst)) (cons (list-map func (car lst)) (list-map func (cdr lst)))]
```

```
    [else (cons (func (car lst)) (list-map func (cdr lst))))])
```

8. `list-reduce`: Reducere stânga pe o listă de liste utilizând o funcție de agregare.

```
(define (list-reduce func lst)
```

```
  (if (null? lst)
```

```
      (error "Cannot reduce an empty list")
```

```
      (let loop ([result (car lst)] [rest (cdr lst)])
```

```
        (if (null? rest)
```

```
          result
```

```
          (loop (func result (car rest)) (cdr rest))))))
```

9. `list-count`: Numără de câte ori un element apare într-o listă de liste.

```
(define (list-count item lst)
```

```
  (cond
```

```
    [(null? lst) 0]
```

```
    [(list? (car lst)) (+ (list-count item (car lst)) (list-count item (cdr lst)))]
```

```
    [(equal? item (car lst)) (+ 1 (list-count item (cdr lst)))]
```

```
    [else (list-count item (cdr lst))]))
```

10. `list-replace`: Înlocuiește toate aparițiile unui element într-o listă de liste cu un alt element.

```
(define (list-replace old new lst)
```

```
(cond
  [(null? lst) '()]
  [((list? (car lst)) (cons (list-replace old new (car lst)) (list-replace old new (cdr lst))))]
  [(equal? old (car lst)) (cons new (list-replace old new (cdr lst)))]
  [else (cons (car lst) (list-replace old new (cdr lst)))]))
```

11. 'list-depth': Calculează adâncimea maximă a unei liste de liste.

```
(define (list-depth lst)
```

```
(cond
```

```
  [(null? lst) 0]
  [((list? (car lst)) (+ 1 (list-depth (car lst))))]
  [else (list-depth (cdr lst))]))
```

12. 'list-sublist': Extrage o sublistă dintr-o listă de liste bazată pe un interval dat.

```
(define (list-sublist start end lst)
```

```
(cond
```

```
  [(null? lst) '()]
  [(and (>= start 0) (< end (length (car lst))))]
  [((cons (sublist (car lst) start end) (list-sublist start end (cdr lst))))]
  [else (list-sublist start end (cdr lst))]))
```

13. 'list-rotate': Rotire ciclică la stânga a elementelor dintr-o listă de liste.

```
(define (list-rotate lst)
```

```
(cond
```

```
  [(null? lst) '()]
  [((list? (car lst)) (cons (list-rotate (cdr lst)) (list-rotate (car lst))))]
  [else (cons (car lst) (list-rotate (cdr lst)))]))
```

14. 'list-interleave': Interlevează două liste de liste.

```
(define (list-interleave lst1 lst2)
```

```
(if (or (null? lst1) (null? lst2))
```

```
'()
```

```
(append (list (car lst1) (car lst2)) (list-interleave (cdr lst1) (cdr lst2)))))
```



1. Funcția `map`: Aplică o funcție dată fiecărui element al unei liste și returnează rezultatele într-o nouă listă.

map :: (a -> b) -> [a] -> [b]

map \_ [] = []

map f (x:xs) = f x : map f xs

2. Funcția `filter`: Selectează elementele dintr-o listă care satisfac o anumită condiție și le returnează într-o nouă listă.

filter :: (a -> Bool) -> [a] -> [a]

filter \_ [] = []

filter p (x:xs) | p x = x : filter p xs

| otherwise = filter p xs

3. Funcția `foldl` (left fold): Aplică o funcție binară asupra elementelor unei liste, folosind un acumulator și parcurgerea listei de la stânga la dreapta.

foldl :: (b -> a -> b) -> b -> [a] -> b

foldl \_ acc [] = acc

foldl f acc (x:xs) = foldl f (f acc x) xs

4. Funcția `foldr` (right fold): Aplică o funcție binară asupra elementelor unei liste, folosind un acumulator și parcurgerea listei de la dreapta la stânga.

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr \_ acc [] = acc

foldr f acc (x:xs) = f x (foldr f acc xs)

5. Funcția `zip`: Preia două liste și le combină într-o singură listă de perechi, conținând elementele corespondente.

zip :: [a] -> [b] -> [(a, b)]

zip [] \_ = []

zip \_ [] = []

zip (x:xs) (y:ys) = (x, y) : zip xs ys

6. Funcția `concat`: Concatenează o listă de liste într-o singură listă.

concat :: [[a]] -> [a]

concat [] = []

concat (x:xs) = x ++ concat xs

7. Funcția `length`: Returnează lungimea unei liste.

length :: [a] -> Int

length [] = 0

length (\_:xs) = 1 + length xs

8. Funcția `reverse`: Inversează ordinea elementelor unei liste.

reverse :: [a] -> [a]

reverse [] = []

reverse (x:xs) = reverse xs ++ [x]

9. Funcția `take`: Preia primele n elemente dintr-o listă.

take :: Int -> [a] -> [a]

take \_ [] = []

take n (x:xs)

| n <= 0 = []

| otherwise = x : take (n-1) xs

10. Funcția `drop`: Elimină primele n elemente dintr-o listă.

drop :: Int -> [a] -> [a]

drop \_ [] = []

drop n lst@(x:xs)

| n <= 0 = lst

| otherwise = drop (n-1) xs

11. Funcția `maximum`: Returnează valoarea maximă dintr-o listă.

maximum :: (Ord a) => [a] -> a

maximum [] = error "Lista este goală."

maximum [x] = x

maximum (x:xs) = max x (maximum xs)

12. Funcția `minimum`: Returnează valoarea minimă dintr-o listă.

minimum :: (Ord a) => [a] -> a

minimum [] = error "Lista este goală."

minimum [x] = x

minimum (x:xs) = min x (minimum xs)

13. Funcția `elem`: Verifică dacă un element există într-o listă.

elem :: (Eq a) => a -> [a] -> Bool

elem [] = False

elem x (y:ys) = x == y || elem x ys

14. Funcția `notElem`: Verifică dacă un element nu există într-o listă.

notElem :: (Eq a) => a -> [a] -> Bool

notElem [] = True

notElem x (y:ys) = x /= y && notElem x ys

15. Funcția `null`: Verifică dacă o listă este goală.

null :: [a] -> Bool

null [] = True

null (\_:\_ ) = False

16. Funcția `head`: Returnează primul element dintr-o listă.

head :: [a] -> a

head [] = error "Lista este goală."

head (x:\_ ) = x

17. Funcția `tail`: Elimină primul element dintr-o listă și returnează restul.

tail :: [a] -> [a]

tail [] = error "Lista este goală."

tail (\_:xs) = xs

18. Funcția `init`: Elimină ultimul element dintr-o listă și returnează restul.

init :: [a] -> [a]

init [] = error "Lista

este goală."

init [\_] = []

init (x:xs) = x : init xs

19. Funcția `last`: Returnează ultimul element dintr-o listă.

```
last :: [a] -> a
last []   = error "Lista este goală."
last [x]  = x
last (_:xs) = last xs
```

20. Funcția `replicate`: Creează o listă cu n copii ale unui element dat.

```
replicate :: Int -> a -> [a]
replicate n x
| n <= 0  = []
| otherwise = x : replicate (n-1) x
```

21. Funcția `zipWith`: Preia două liste și aplică o funcție binară asupra elementelor corespondente.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _      = []
zipWith _ _ []      = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

22. Funcția `any`: Verifică dacă orice element dintr-o listă satisfac o anumită condiție.

```
any :: (a -> Bool) -> [a] -> Bool
any _ []    = False
any p (x:xs) = p x || any p xs
```

23. Funcția `all`: Verifică dacă toate elementele dintr-o listă satisfac o anumită condiție.

```
all :: (a -> Bool) -> [a] -> Bool
all _ []    = True
all p (x:xs) = p x && all p xs
```

24. Funcția `splitAt`: Desparte o listă în două la un anumit index.

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt _ []    = ([], [])
splitAt n xs
| n <= 0  = ([], xs)
| otherwise = let (ys, zs) = splitAt (n-1) (tail xs) in (head xs : ys, zs)
```

25. Funcția `concatMap`: Map-ează o funcție asupra elementelor unei liste și apoi concatenează rezultatele.

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

```
concatMap [] = []
```

```
concatMap f (x:xs) = f x ++ concatMap f xs
```

1. **Exercițiul:** Implementați o funcție `subsets :: [a] -> [[a]]` care primește o listă și returnează toate submulțimile acelei liste.

```
subsets :: [a] -> [[a]]
```

```
subsets [] = [[]]
```

```
subsets (x:xs) = subsets xs ++ map (x:) (subsets xs)
```

2. **Exercițiul:** Implementați o funcție `permutations :: [a] -> [[a]]` care primește o listă și returnează toate permutările posibile ale elementelor listei.

```
permutations :: [a] -> [[a]]
```

```
permutations [] = [[]]
```

```
permutations (x:xs) = concatMap (insertElem x) (permutations xs)
```

where

```
insertElem :: a -> [a] -> [[a]]
```

```
insertElem x [] = [[x]]
```

```
insertElem x (y:ys) = (x:y:ys) : map (y:) (insertElem x ys)
```

3. **Exercițiul:** Implementați o funcție `combinations :: Int -> [a] -> [[a]]` care primește un număr `k` și o listă și returnează toate combinațiile de `k` elemente posibile din lista dată.

```
combinations :: Int -> [a] -> [[a]]
```

```
combinations 0 [] = [[]]
```

```
combinations k xs = [x:ys | x:xs' <- tails xs, ys <- combinations (k-1) xs']
```



6. Instanțiați clasa `Ord` pentru funcții Haskell care iau un argument numeric, astfel încât o funcție este "mai mică" decât alta dacă valoarea ei este mai mică decât a celeilalte funcții **în cel puțin unul** dintre numerele întregi între 1 și 10.

*Soluție:*

```
instance (Num a, Enum a, Ord b) => Ord (a -> b) where
    f <= g = or (zipWith (<=) (map f [1..10]) (map g [1..10]))
```

Notă: `Enum a` nu era cerut.

6. Evidențiați o posibilă instanță a clasei Haskell de mai jos: *Soluție:*

```
class MyClass c where
    f :: c a -> a
```

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

6. Scrieți o instanță posibilă a clasei de mai jos, conținând o implementare **neconstantă** a funcției `f`.

```
class MyClass c where
    f :: Num a => c a -> c a -> c a
```

*Soluție:*

```
instance MyClass Maybe where
    f (Just x) (Just y) = Just $ x + y
    f _ _ = Nothing
```

6. Supraîncărcați în Haskell operatorul de comparație pe liste, astfel încât o listă să fie mai mică sau egală cu alta, dacă toate elementele din prima listă sunt mai mici sau egale cu toate elementele din a doua. Spre exemplu,  $[2, 3, 1] \leq [5, 4, 3]$ , dar nu avem că  $[2, 4] \leq [3, 5]$ .

*Soluție:*

```
instance Ord a => Ord [a] where
    xs <= ys = maximum xs <= minimum ys
```

6. Scrieți o instanță posibilă a clasei de mai jos, conținând o implementare **neconstantă** a funcției `f`.

```
class MyClass c where
    f :: Ord a => c a -> c a -> c Bool
```

*Soluție:*

```
instance MyClass Maybe where
    f (Just x) (Just y) = Just $ x <= y
    f _ _ = Nothing
```

6. Supraîncărcați în Haskell operatorii `(+)` și `(*)` pentru valori booleene, pentru a surprinde operațiile de *sau*, respectiv *și* logic.

*Soluție:*

```
instance Num Bool where
    (+) = (||)
    (*) = (&&)
```

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.

Instanțiați această clasă pentru tipul `data Triple a = T a a a`

*Soluție:*

```
class Ended t where frontEnd :: t v -> v; backEnd :: t v -> v
instance Ended Triple where

    frontEnd (T x _ _) = x
    backEnd (T _ _ x) = x
```

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.

Instantiați această clasă pentru tipul `data NestedL a = A a | L [NestedL a]`

*Soluție:*

```
class Ended t where frontEnd :: t v -> v; backEnd :: t v -> v
instance Ended NestedL where

    frontEnd (A a) = a; frontEnd (L l) = frontEnd $ head l
    backEnd (A a) = a; backEnd (L l) = backEnd $ last l
```

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.

Instantiați această clasă pentru tipul `data Pair a = MakePair a a`

*Soluție:*

```
class Ended t where frontEnd :: t v -> v; backEnd :: t v -> v
instance Ended Pair where
```

```
    frontEnd (MakePair x _) = x
    backEnd (_ x) = x
```

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.

Instantiați această clasă pentru tipul listă Haskell.

*Soluție:*

```
class Ended t where frontEnd :: t v -> v; backEnd :: t v -> v
instance Ended [] where
```

```
    frontEnd = head
    backEnd = last sau head . reverse
```

5. Instantiați clasa `Show` pentru funcții Haskell care iau un argument numeric, astfel încât afișarea unei funcții `f` va produce afișarea rezultatelor aplicării funcției pe numerele de la 1 la 10. E.g. afișarea lui `(+1)` va produce: `234567891011`.

*Soluție:*

```
-# LANGUAGE FlexibleInstances #- -- nu este cerut în rezolvarea din examen
instance (Enum a, Num a, Show b) => Show (a -> b) where
    show f = concatMap (show . f) [1..10]
-- Enum nu este cerut în rezolvarea din examen
```

5. Instantiați clasa `Show` pentru funcții Haskell care iau un argument numeric, astfel încât afișarea unei funcții `f` va produce afișarea rezultatelor aplicării funcției pe numerele de la 1 la 10. E.g. afișarea lui `(+1)` va produce: `234567891011`.

*Soluție:*

```
-# LANGUAGE FlexibleInstances #- -- nu este cerut în rezolvarea din examen
instance (Enum a, Num a, Show b) => Show (a -> b) where
    show f = concatMap (show . f) [1..10]
-- Enum nu este cerut în rezolvarea din examen
```

6. Instantiați în Haskell clasa `Eq` pentru tripluri, considerând că `(a1, a2, a3)` este egal cu `(b1, b2, b3)` dacă `a1 == b1` și `a2 == b2`.

*Soluție:*

```
instance (Eq a, Eq b) => Eq (a, b, c) where (a1, a2, _) == (b1, b2, _) = (a1 == b1)
&& (a2 == b2)
```

6. Instanțiați în Haskell clasa `Ord` pentru tripluri (știind că `Eq` este deja instanțiată), considerând că  $(a_1, a_2, a_3)$  este mai mic decât  $(b_1, b_2, b_3)$  dacă  $a_1 < b_1$ .

*Soluție:*

```
instance Ord a => Ord (a, b, c) where (a1, _, _) < (b1, _, _) = a1 < b1
```

5. Instanțiați în Haskell clasa `Ord` pentru perechi. Ordinea perechilor va fi dată de compararea celui de-al doilea element din pereche. E.g.  $(1, 2) > (2, 0)$  (pentru că  $2 > 0$ ).

*Solutie:*

NOTĂ: Pentru ca implementarea să compileze am folosit aici `MyOrd` și `#<=` în loc de `Ord` și `<=`, definite astfel: `class Eq a => MyOrd a where (#<=) :: a -> a -> Bool.`

Soluția cerută, (dar cu `Ord` și `<=` în loc de `MyOrd` și `#<=`), era:

```
instance (Eq a, Ord b) => MyOrd (a, b) where  
    (_, y1) #<= (_, y2) = y1 <= y2
```

5. Instanțiați în Haskell clasa `Ord` pentru liste. Ordonarea listelor va fi dată de compararea primului element din fiecare listă. E.g.  $[1, 2, 3] < [2, 3, 4]$  pentru că  $1 < 2$ .

*Soluție:*

NOTĂ: Pentru ca implementarea să compileze am folosit aici `MyOrd` și `#<=` în loc de `Ord` și `<=`, definite astfel: `class Eq a => MyOrd a where (#<=) :: a -> a -> Bool.`

Soluția cerută, (dar cu `Ord` și `<=` în loc de `MyOrd` și `#<=`), era:

```
instance Ord a => MyOrd [a] where  
    (h1:_ ) #<= (h2:_ ) = h1 <= h2
```

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  $1, \dots, 100$ .

*Soluție:*

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

7. Supraîncărcați în Haskell afișarea funcțiilor care au ca parametru un număr, afișându-se valoarea funcției în punctul 0.

*Solutie:*

```
instance (Show b, Num a) => Show (a -> b) where show f = show (f 0)
```

7. Supraîncărcați în Haskell ordonarea funcțiilor care au ca parametru un număr, ordonând funcțiile după valoarea lor pentru argumentul 0. Se consideră că operatorul de egalitate între astfel de funcții a fost deja definit.

*Soluție:*

```
instance (Num a, Ord b) => Ord (a -> b) where f > g = f 0 > g 0
```

5. Supraîncărcați în Haskell afișarea funcțiilor care au ca parametru un număr, afișându-se valoarea funcției în punctul 1.

*Solutie:*

```
instance (Show b, Num a) => Show (a -> b) where show f = show (f 1)
```



## Pregátilne etapy

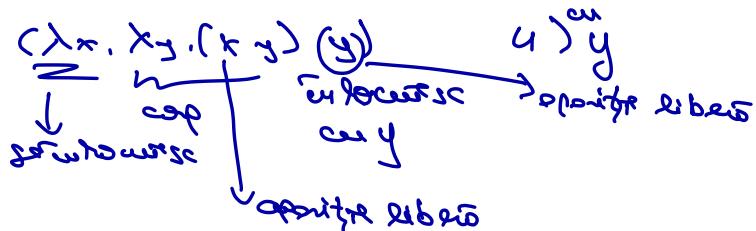
## F (Accel LAMBDA)

## 1 Colored Symbols

- a)  $\ell^1$ ) fb so unlosen q. libe

2) alle verordnet X

3) Ein exp. hy. (x y)



- b)  $\beta$ -reducção:  $\lambda y. (y \ y)$  é trivialmente reduzível ao  $\lambda y. y$ .

fb55 aplic o conversão pf o exito als

$$(\lambda x. \lambda y. (x \neq y) y) \xrightarrow{\alpha} \lambda x. \lambda z. (x \neq z) y \xrightarrow{\beta} \lambda z. (x \neq z) [y/x] = \lambda z. (y \neq z)$$

↑  
Free var. zip. bsb.

$$2. (\lambda x. (\underset{x}{\underset{\beta}{=}} \underset{m}{\underset{\beta}{=}} x) x) a) \xrightarrow{\beta} (\underset{x}{\underset{\beta}{=}} \underset{m}{\underset{\beta}{=}} a) \xrightarrow{\beta} x_{(a/x)} \rightarrow a$$

do b6 ch b6 st. ↑

$$\text{debst\,b\,m: } (\lambda x. (\lambda x. x \, x) \, a) \xrightarrow{\beta} (\lambda x. x \, a) \rightarrow a.$$

of course the parents will, so out of their own interest they → must modify nearly all our new laws if possible.

$$\xrightarrow{\beta} (\overbrace{\lambda x. \lambda y. \underline{x}}^{\equiv} (\lambda x. x) \underline{z}) \rightarrow (\overbrace{\lambda y. \lambda x. x}^{\equiv} \underline{z}) \rightarrow \lambda x. x$$

www.ouroplib.

$$4. (\lambda y. ((\lambda x. \lambda y. x(y)) z) \lambda x. \lambda y. y) \xrightarrow{=} ((\lambda x. \lambda y. x, \lambda x. \lambda y. y) z) \xrightarrow{=} (\lambda y. \lambda x. \lambda y. y z) \xrightarrow{=}$$

$$5. (\underbrace{(\lambda x. \lambda y. \lambda z. y \underbrace{\lambda r. r})}_{\text{a}} (\lambda z. \lambda t. z z)) z \downarrow_B \text{ liber } \uparrow \rightarrow \lambda x. \lambda y. y \rightarrow \text{def of } \lambda \text{ w/ } \text{curry} \text{ w/ } \text{2nd const at } 2 - \text{pos - 1 in right M}$$

$$\text{new opp side } \rightarrow \text{new } x \Rightarrow ((\lambda y. x^2. y) (\lambda z. \lambda t. z^2)) \circ_2 \rightarrow$$

$$\underset{=}{\lambda} y. \lambda x. y \underbrace{(x^2, x+2, 2)}_2$$

$$\rightarrow_B (\lambda w. \underbrace{(\lambda z. \lambda t. z - z)}_{\text{dub.}} - z) \rightarrow (\lambda z. \lambda t. \underbrace{\hat{z} - z}_{\text{dub.}}) \xrightarrow[\text{dub.}]{B} (\lambda t. z - z)$$

$$6. \circ := \lambda f. \lambda x. x \quad \text{succ} := \lambda n. \lambda f. \lambda x. (f((n\ f) x))$$

(succ  $\circ$ )!

$$(\underbrace{\lambda n. \lambda f. \lambda x. (f((n\ f) x))}_{\text{succ}} \underbrace{\lambda f. \lambda x. x}_{\circ}) \rightarrow \lambda f. \lambda x. (f((\underbrace{\lambda f. \lambda x. x}_{\text{succ}} \underbrace{f}_{\text{succ}}) x)) \rightarrow$$

↳ no need for so many terms

$$\rightarrow \lambda f. \lambda x. (f(\underbrace{\lambda x. x}_{\text{succ}})) \rightarrow \lambda f. \lambda x. (f x) \text{ now we can use -i need for}$$

$$7. \text{true} = \lambda x. y. x$$

$$\text{false} = \lambda x. y. y$$

$$\lambda x. y. (\text{true} \ y \ \text{true})?$$

$$\text{true} = \lambda x. \lambda y. x$$

$$\text{false} = \lambda x. \lambda y. y$$

$$\text{op} = \lambda x. \lambda y. (x \ y \ \text{true}, \lambda x. \lambda y. x)$$

$\text{NOT} \rightarrow$  mon, neither binary

or split the task into AND, OR per component.

$$(\text{op true } y) \Rightarrow (\underbrace{\lambda x. \lambda y. (\overbrace{x \ y \ \text{true}}^{\text{def true}})}_{\text{split}} \ \overbrace{\text{true } y}^{\text{true}}) \rightarrow (\text{true } y \ \text{true}) \rightarrow$$

put one in dom of  
true in def true, den  
→ false find it  
numbers in den.

so for every argument  
either fi op true or  
y & true

$$\rightarrow (\underbrace{\lambda x. \lambda y. x \ y \ \text{true}}_{\text{def true}}) \rightarrow y$$

$$(\text{op false } y) \rightarrow (\lambda x. \lambda y. (x \ y \ \text{true}) \text{false } y) \rightarrow (\text{false } y \ \text{true}) \rightarrow$$

$$\rightarrow (\lambda x. \lambda y. y \ y \ \text{true}) \rightarrow \text{true}$$

$$\text{op} \leq \text{=} \rightarrow p \Rightarrow \text{op} \leq \gamma \ell \vee \text{op}$$

$$\begin{array}{l} \text{if } \text{false} \vee y = y \\ \text{if } \text{true} \vee y = \text{true} \end{array}$$

# LAMBDA

2022-a

$$(\lambda y. \lambda x. (\underline{y} \ u) (x \ x))$$

lambdas y lambdas x aplican peste y aplican peste x (aplican peste x).

coti ...? coti beta-reduz?  $\rightarrow$  o que é o que x  $\Rightarrow$  reduzir primo chisto  
++ ( $\lambda y$  lo que é isto)

de la esquio more ptes teos exp este singular bizarro

$$\text{exp: } \lambda x. (\underline{y} \ u)$$

o e de ptes primas ptes exp

ptes formal: y

ptes formal e ptes primas  $\lambda$

ptes actua: x aplicar pte x (x x) ptos act e celo d. dito  
++ ce je do-

intencionar exp; destruir & transformar exp pte y em ptos act.

$\hookrightarrow$  nra e o ido bnd ptes et. com pte (x x) unde x e nra libra em  
lado direito, no q sub cap d nra fct ova os ptos formal x  $\Rightarrow$   
 $\Rightarrow$  o q x em fr ligado x de los  $\lambda x$

$$\begin{aligned} &\Downarrow \\ &\text{nra fcs dlo conv. nra d red. pte x: } \lambda x. (\underline{y} \ u) \rightarrow \lambda z. (\underline{y} \ u) \\ &\Rightarrow \lambda z. (\underline{y} \ u) [(\underline{x} \ x) / \underline{y}] \end{aligned}$$

$$\begin{aligned} &\Downarrow \\ &\text{pte y d'inde em (x x)} \\ &\Rightarrow \lambda z. ((\underline{x} \ x) (\underline{z} \ z)) \rightarrow \text{resultado 1st fcs.} \end{aligned}$$

ou vam p redet? - NU.  $\Rightarrow$  ou termina !!

2019-B

$$E = (\underline{y} (\lambda x. \lambda x. x (\underline{y} \ u)))$$

ou 3 B-reduti. Nra em cel nra d'inde d'apto  $\Rightarrow$   
 $\Rightarrow$  fct identit apl pte y  $\Rightarrow$   $(\underline{y} (\lambda x. \lambda x. x \ u))$

pte d conv.  $\Rightarrow$   $(\underline{y} (\lambda z. \lambda x. x \ u)) \Rightarrow (\underline{y} \lambda x. x)$  nra d ptes et. cont.  
nra d ptes et.  $\Rightarrow$   $(\underline{y} \lambda x. x)$  nra d ptes et. cont. sd!  
nra d ptes et.  $\Rightarrow$  scpd de ()

$$\begin{aligned} &\text{2018-B} \\ &\lambda x. \lambda y. ((\underline{\lambda x. \lambda y. (\underline{x} \ u)}) (\underline{x} \ u)) \rightarrow \lambda x. \lambda y. ((\lambda x. \lambda z. x (\underline{y} \ z)) (\underline{x} \ u)) \rightarrow \\ &\beta \end{aligned}$$

decis =  $\rightarrow$  co decisi;  $\cup$  co fundatori;  $\hookrightarrow$  em co fundatori;  $\sqsubset$  = apj. segui  
ptos formal ptos formal ptos actua

)

dois sent  
depois sou  
data em lego.  
camino

Reduceri expresia  $\lambda$ : 2022-IV

$$(\lambda x. (\lambda x. (x \ x) \lambda x. x) \ x) = (\lambda x. \lambda x. x \ (x \ x)) \equiv$$

Sent 2 β redcări  $\Rightarrow$  Îl luăm pe cel din interior:

$$(\lambda x. (x \ x) \lambda x. x) = (x \ x)_{[x/x]} = (\lambda x. x \ \lambda x. x) =$$

param act =  $\lambda x. x$  = function identity  $= \lambda x. x$

param formal x (de la paramul  $\lambda x$ ) o lăsat  
ofă

corp (x x)

$$\text{trebuie să redemulsesc: } (\lambda z. \lambda x. x \ (z \ z)) = (z \ z) \quad ?? \quad \text{dvs.}$$

$$\text{param act} = z$$

$$\text{param form.}(z \ z)$$

$$\text{corp } \lambda x. x \rightarrow \text{identity}$$

folg A

$$E = (\lambda x. (x (\lambda y. z \ x)) \ \lambda x x) = (\lambda x. (x z) \ \lambda x. x) \equiv$$

Sent 2 β redcări. Începem cu cel mai din spate:

$$(\lambda y. z \ x) = z_{[x/y]} = z$$

corp: z

param act: x

param formal: y

$$\Rightarrow \alpha = (\lambda x. (x z) \ \lambda y. y) \equiv$$

corp: (x z)

param act:  $\lambda y. y$

param formal: x

$$\Rightarrow (x z)_{[\lambda y. y / x]} = (\lambda y. y z) \equiv z$$

```
{-
  1. Să se găsească cuvintele care au lungimea cel puțin egală
  cu 10 caractere în două moduri:
    1) folosind filter
    2) folosind list comprehensions
-}
```

```
findStringsLongerThanTenChars :: [String] -> [String]
findStringsLongerThanTenChars l = filter (\x -> length x >= 10) l
```

```
findStringsLongerThanTenChars2 :: [String] -> [String]
findStringsLongerThanTenChars2 l = [x | x <- l, length x >= 10]
```

```
{-
  2. Să se construiască o listă de perechi de tip (string,
  lungime_string) în două moduri:
    1) folosind map
    2) folosind list comprehensions
-}
```

```
buildPairsStringLength :: [String] -> [(String, Int)]
buildPairsStringLength l = map (\x -> (x, length x)) l
```

```
buildPairsStringLength2 :: [String] -> [(String, Int)]
buildPairsStringLength2 l = [(x, length x) | x <- l]
```

```
{-
  3. Implementați, folosind obligatoriu list-comprehensions,
  operații pe multimi:
    intersecție, diferență, produs cartezian. Utilizați ulterior
  funcțiile definite anterior
    pentru a reprezenta reuniunea multimilor.
-}
```

```
setIntersection :: Eq a => [a] -> [a] -> [a]
setIntersection a b = [x | x <- a, x `elem` b]
```

```
setDiff :: Eq a => [a] -> [a] -> [a]
setDiff a b = [x | x <- a, x `notElem` b]
```

```
cartProduct :: [a] -> [b] -> [(a, b)]
cartProduct a b = [(x, y) | x <- a, y <- b]
```

```
setUnion :: Eq a => [a] -> [a] -> [a]
setUnion a b = a ++ setDiff b (setIntersection a b)
```

```
naturals = [0..]
```

```
naturals = iter 0
  where iter x = x : iter (x + 1)
```

```

> :t iterate
iterate :: (a -> a) -> a -> [a]

naturals = iterate (\x -> x + 1) 0 -- SAU
naturals = iterate (+ 1) 0

ones = repeat 1 -- [1, 1, 1, ...]
onesTwos = intersperse 2 ones -- [1, 2, 1, 2, ...]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs) -- sirul lui Fibonacci
powsOfTwo = iterate (* 2) 1 -- puterile lui 2
palindromes = filter isPalindrome [0..] -- palindroame
    where
        isPalindrome x = show x == reverse (show x) -- truc: reprezint
numarul ca String

f $ x = f x

length $ 3 : [1, 2] -- length (3 : [1, 2])

sum xs = foldl (+) 0 xs

sum = foldl (+) 0

(.) :: (b -> c) -> (a -> b) -> a -> c

> length (tail (zip [1,2,3,4] ("abc" ++ "d")))
> length $ tail $ zip [1,2,3,4] $ "abc" ++ "d"

> (length . tail . zip [1,2,3,4]) ("abc" ++ "d")
> length . tail . zip [1,2,3,4] $ "abc" ++ "d"

flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

> :t map
map :: (a -> b) -> [a] -> [b]
> :t flip map
flip map :: [a] -> (a -> b) -> [b]

myIntersperse :: a -> [a] -> [a]
myIntersperse y = foldr (++) [] . map (: [y])

```

Sintaxa if:

```
Haskell
if a < 0 then
    if (a > 10)
        then a * a
        else 0
    else -1
```

Definirea unei funcții:

```
Haskell
-- cu `if .. else .. then`
sumList :: [Int] -> Int
sumList l = if null l then 0 else head l + sumList (tail l)

-- cu găzzi
sumList3 :: [Int] -> Int
sumList3 l
| null l = 0
| otherwise = head l + sumList3 (tail l)

-- cu `case .. of`
sumList4 :: [Int] -> Int
sumList4 l = case l of
[] -> 0
(x:xl) -> x + sumList4 xl

-- cu pattern matching (sintactic sugar pentru varianta cu `case` de mai sus)
sumList2 :: [Int] -> Int
sumList2 [] = 0
sumList2 (x:xl) = x + sumList2 xl
```

Funcționale:

```
-- map
map (\x -> x + 1) [1, 2, 3, 4] -- [2, 3, 4, 5]
map (+ 1) [1, 2, 3, 4] -- [2, 3, 4, 5]

-- filter
filter (\x -> mod x 2 == 0) [1, 2, 3, 3, 4, 5, 6] -- [2, 4, 6]

-- foldl
reverse $ foldl (\acc x -> x : acc) [] [1, 2, 3, 4, 5] -- [1, 2, 3, 4, 5]

-- foldr
foldr (\x acc -> x : acc) [] [1, 2, 3, 4, 5] -- [1, 2, 3, 4, 5]
```

Observăm că, spre deosebire de Racket, în Haskell funcționalele foldl și foldr primesc funcții cu semnături diferite:

foldl :: (b -> a -> b) -> b -> [a] -> b (funcția ajutătoare primește acumulatorul și apoi elementul curent)

`foldr :: (a -> b -> b) -> b -> [a] -> b` (funcția ajutătoare primește elementul curent și apoi accumulatorul)

Legări:

```
-- cu let
f a =
  let c = a
    b = a + 1
  in (c + b) -- let din Racket

g a =
  let c = a
    b = c + 1
  in (c + b) -- let* din Racket

h a =
  let c = b
    b = a + 1
  in (c + b) -- letrec din Racket, aici nu avem eroare datorită
evaluării leneșe

-- cu where
f' a = (c + b)
  where
    c = a
    b = a + 1 -- let din Racket

g' a = (c + b)
  where
    c = a
    b = c + 1 -- let* din Racket

h' a = (c + b)
  where
    c = b
    b = a + 1 -- letrec din Racket, aici nu avem eroare
datorită evaluării leneșe
```

$$f \cdot x \cdot y = (\text{head } x, \text{head } y) : f(\text{tail } x)(\text{tail } y)$$

$\Rightarrow$  function  $f(x, y)$ ;  $x, y$  are facts

$$f = \lambda x \cdot y \rightarrow (\text{head } x, \text{head } y)$$

este symbolo de  $x$  y cont. aio  $\uparrow$  & este expr 1)

$$\text{expr 2} \leftarrow (\text{head } x, \text{head } y)$$

$$: \text{cons} = \text{expr 3}$$

$$f(\text{tail } x)(\text{tail } y) = \text{expr 4} \quad (f \text{ explicit for } \dots)$$

pti expr 1) tipul: dacs  $x$  e de tipul  $a$  &  $y$  e de tipul  $b$  &  $\text{expr}(3)$  e de tipul  $c$

$$\Rightarrow f \text{ e de tipul } a \text{ in } a \text{ & } b \text{ in } b \Leftrightarrow x :: a \Rightarrow f :: a \rightsquigarrow b$$

(1):  $\text{head } x \Rightarrow x \text{ listo} \Rightarrow x :: [d] \quad (3) :: c \quad y :: [e] \quad (2) :: (d, e)$   
 $\Rightarrow f \text{ e de tipul } 2 \text{ element, num pe lista}$

(4) let liste de tipul  $d \Delta e \Rightarrow$  dim expr 4  $\Rightarrow f :: [d] \rightarrow [e] \rightarrow c \quad (4) :: c$

(3)  $\Rightarrow$  cons  $\Rightarrow$  dacs nu e nif to ss fr a lists de cu ce sa peste for cons

nu e dacs nu (head din lista) to ss fr new expr 2 deu procedur  $\Rightarrow$

$$\Rightarrow (3) : c = [(d, e)] \quad \text{fieci pos cu pos,}$$

$$\Rightarrow f :: [d] \rightarrow [e] \rightarrow [(d, e)] \quad \text{impoziti in expresie}$$

primul  $x, y$ , le defineste pozitii de  $x, y$  & cu  $\text{head } x \rightsquigarrow$   $x$  si  $\text{head } y \rightsquigarrow$   $y$   
 $\Rightarrow$  semnificatii de zip din Haskell sau zap with in const puncte

$$f \cdot x \cdot y = \text{if } (x = \text{head } y) \text{ then } (x) \text{ else } f \cdot x \cdot (\text{tail } y)$$

$\hookrightarrow$  cum fel de filter

$$\text{well-typed cond e } x = \text{head } y \Rightarrow y \text{ e listo}$$

$$= = \rightarrow \text{acest tip? son dependent de ce e head } F_2?$$

$\downarrow$

$$\begin{cases} x :: a \\ y :: [b] \text{ dae } \text{head } y = x \Rightarrow b = a \Rightarrow y :: [a] \\ \text{int } [x] \Rightarrow [a] \end{cases}$$

$F_2$  e un element  
nu e absolut n'tip

$$f :: F_2 a \Rightarrow a \rightarrow [a] \rightarrow [a]$$

Wig- $\lambda$  f x y = x y (y x)

neut ist so f als def pt f

f x y:  $\lambda$ -term  $\lambda$ :  $f = \lambda x y \rightarrow x y (y x)$

$x :: a, y :: b; (y x) :: c \Rightarrow [c = j]$

$$x \text{ der } y \Rightarrow \frac{x :: \text{der} \quad y :: g \rightarrow b}{(x y) :: e}$$

$$y \text{ per } x: \frac{y :: i \rightarrow f \quad x :: k \rightarrow l}{(y x) :: j}$$

$$\frac{a :: b \quad b :: b}{x y :: a \rightarrow b} \\ \boxed{a \rightarrow b = e}$$

$$x y \text{ per } f(x): \frac{(x y) :: m \rightarrow n \quad (y x) :: o \rightarrow p}{(x y (y x)) :: n}$$

$$x y (y x) :: n \quad f :: a \rightarrow b \rightarrow n$$

Wig-b f = map (++)

map f  $L \rightarrow L'$   $\Rightarrow$  map ::  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$  (1)

(++) :  $L_1 + + L_2 \rightarrow L_3 \Rightarrow (++) :: [c] \rightarrow [c] \rightarrow [c]$   
 $L_1 \quad L_2 \quad L_3 = L_1 \cup L_2$  (2)

così map (1) e (2) hanno lo stesso tipo

map ::  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

def co (++) ::  $[c] \xrightarrow{a} ([c] \xrightarrow{b} [c]) \Rightarrow a = [c]$

$$b = [c] \rightarrow [c]$$

$\Rightarrow f :: [[c]] \rightarrow [[c] \rightarrow [c]]$

map (++)  $\rightarrow$  lista appende la lista di liste  $\Rightarrow$  prima lista

lista de liste d'insieme tot a lists de liste.

W18-a

$$f \circ g = \underbrace{x \circ y}_{x \in a} \circ z = z \text{ comp w/ } y$$

$x \in a$  does  $xy \in \text{fact} \& z \in \text{funct}$

$$\left. \begin{array}{l} y \in b \\ z \in c = d \rightarrow e \\ xy \in f \rightarrow h \\ \text{does } z \circ (xy) \Rightarrow e = g \end{array} \right\} \Rightarrow x : b \rightarrow e \rightarrow h = a$$

$f : (b \rightarrow e \rightarrow h) \xrightarrow{x} b \xrightarrow{y} (d \rightarrow e) \xrightarrow{z} d \rightarrow e.$

$\underline{d \rightarrow e / \rightarrow g \in h}$

W18-b

$$f \circ g = x \circ y \circ z$$

Partial Input:

$$x : a ; y : b ; z : c$$

now we see  $x$  is const since  $x$  to  $a$  for fact, so fact  $y \circ z$  const for compus  $\Rightarrow$

$$\Rightarrow x : d \rightarrow e \stackrel{a}{=} ; \boxed{y \circ z : f \rightarrow h \Rightarrow \boxed{y : c \rightarrow g \rightarrow h} = b}$$

$$\text{does } y : b \quad x \circ y \circ z \Rightarrow x \circ y z \Rightarrow x (y z (\dots))$$

$\boxed{z : c}$

$\cup_{x : h \rightarrow e}$

$$f : (h \rightarrow e) \rightarrow (c \rightarrow g \rightarrow h) \xrightarrow{z} c \rightarrow g \rightarrow e$$

$$g \rightarrow h \rightarrow h \rightarrow e$$

$$W18-a \quad f \circ g = (g \circ x) \circ x$$

$$x : a, y : b$$

$$(g \circ x) \Rightarrow y \in \text{fact} \Rightarrow y : \underline{c \rightarrow d} = b$$

$y$  explicit per  $x$ , does  $y \circ x \in \text{fact}$

$$\begin{array}{ccc} (g \circ x) : c \rightarrow d & (y \circ x) : a \rightarrow d & \\ \Downarrow & y \text{ is const w/ } x : a & \Downarrow \\ y : a \rightarrow a \rightarrow d & & (g \circ x) : a \rightarrow d \end{array}$$

$$x : a, y : a \rightarrow a \rightarrow d; \text{ does } d \Rightarrow$$

$$\Rightarrow f : a \rightarrow \underbrace{(a \rightarrow a \rightarrow d)}_y \rightarrow d$$

$\text{does } \underbrace{a \rightarrow d}_n$

W16-b f  $\times$  g = x (y x)

gen:  $f: a \rightarrow b \rightarrow c$ .  
x y rez

x este fact se aplică (y x)  $\Rightarrow$  x :: d  $\rightarrow$  e.

y este fact se aplică pe x  $\Rightarrow$  y :: f  $\rightarrow$  h

x este (y x)  $\Rightarrow$  x primește conținut y  $\Rightarrow$  x :: h  $\Rightarrow$  d = h.

y este x  $\Rightarrow$  y primește conținut x  $\Rightarrow$  y :: e  $\rightarrow$  h  $\Rightarrow$  g = e

x :: h  $\rightarrow$  e; y :: e  $\rightarrow$  h

dici:  $f: a \rightarrow b \rightarrow c$

$x :: d \rightarrow e$   $\nearrow$   $d = h$  (x este valoare a(y))  
 $y :: g \rightarrow h$   $\searrow$   $e = c$  (f este valoare de x.)

$f: (d \rightarrow c) \rightarrow ((d \rightarrow c) \rightarrow d) \xrightarrow{a=g=d \rightarrow e}$  (y este valoare f(x)).  
 $\underbrace{\hspace{1cm}}_{\text{do}}$   $b = g \rightarrow h \equiv g \rightarrow d \Rightarrow b = (d \rightarrow c) \rightarrow d$

y este valoare ex div func  $\Rightarrow \underline{d \rightarrow c}$ .

W15-a

$f x = f (f x)$   
 $f: a$   $\rightarrow$  l nu este compus.

$x :: b$   $f$  este func  $\Rightarrow f: R \rightarrow d$   
 $f$  primește ca arg x  $\Rightarrow x :: e$

f este o func apăsa pe următorul f x  $\Rightarrow f: R \rightarrow R$

$x :: R$ ;  $f: e \rightarrow e$ .

sau:  $f: a \rightarrow b$  explicit  
 $\frac{x :: a}{(f(f x)) :: b}$  mai sp leg  
 $f: c \rightarrow d$   
 $(f x) :: c$   
 $d = b$   
 $f: e \rightarrow g$   
 $x :: e = a$   
 $g = c$   
 $\text{din } f: a \rightarrow c = e, b = d = g.$   
 $f: a \rightarrow a$   $\therefore f: it \rightarrow e$

1. Problema: Scrieți un cod în Racket pentru a găsi toate numerele prime dintr-o listă de numere.

```
(define (prime-numbers lst)
  (filter prime? lst))
(prime-numbers '(2 3 4 5 6 7 8 9 10))
```

2. Problema: Scrieți un cod în Racket pentru a găsi suma tuturor numerelor pare dintr-o listă de numere.

```
(define (sum-even-numbers lst)
  (foldl (lambda (x acc) (if (even? x) (+ x acc) acc)) 0 lst))
(sum-even-numbers '(1 2 3 4 5 6 7 8 9 10))
```

3. Problema: Scrieți un cod în Racket pentru a verifica dacă o listă de numere este palindrom.

```
(define (palindrome? lst)
  (equal? lst (reverse lst)))
(palindrome? '(1 2 3 2 1))
```

4. Problema: Scrieți un cod în Racket pentru a găsi cel mai mare număr dintr-o listă de numere.

```
(define (max-number lst)
  (apply max lst))
(max-number '(5 2 9 1 7 3))
```

5. Problema: Scrieți un cod în Racket pentru a calcula suma cifrelor unui număr dat.

```
(define (sum-of-digits n)
  (apply + (map (lambda (x) (- (char->integer x) (char->integer #\0))) (string->list (number->string n)))))
(sum-of-digits 12345)
```

6. Problema: Scrieți un cod în Racket pentru a verifica dacă un cuvânt este un palindrom.

```
(define (palindrome-word? word)
  (equal? word (reverse word)))
(palindrome-word? "radar")
```

7. Problema: Scrieți un cod în Racket pentru a verifica dacă toate cuvintele dintr-o listă au aceeași lungime.

```
(define (same-length? lst)
```

```
(apply = (map string-length lst))
(same-length? '("abc" "def" "ghi"))
```

8. Problema: Scrieți un cod în Racket pentru a găsi media aritmetică a unei liste de numere.

```
(define (average lst)
  (/ (apply + lst) (length lst)))
(average '(1 2 3 4 5))
```

9. Problema: Scrieți un cod în Racket pentru a găsi produsul tuturor numerelor impare dintr-o listă de numere.

```
(define (product-of-odd-numbers lst)
  (foldl (lambda (x acc) (if (odd? x) (* x acc) acc)) 1 lst))
(product-of-odd-numbers '(1 2 3 4 5))
```

10. Problema: Scrieți un cod în Racket pentru a găsi toate cuvintele dintr-un text dat care încep cu litera "a".

```
(define (words-starting-with-a text)
  (filter (lambda (word) (char=? (string-ref word 0) #\a)) (string-split text)))
(words-starting-with-a "ana are mere si pere")
```

11. Problema: Scrieți un cod în Racket pentru a găsi cel mai mic număr par dintr-o listă de numere.

```
(define (min-even-number lst)
  (apply min (filter even? lst)))
(min-even-number '(3 1 4 1 5 9 2 6 5))
```

12. Problema: Scrieți un cod în Racket pentru a verifica dacă toate cuvintele dintr-o listă sunt scrise cu litere mici.

```
(define (all-lowercase? lst)
  (every (lambda (word) (string=? word (string-downcase word))) lst))
(all-lowercase? '("apple" "banana" "pear"))
```

13. Problema: Scrieți un cod în Racket pentru a găsi numerele prime mai mici decât un număr dat.

```
(define (primes-less-than n)
  (filter prime? (range 2 n)))
```

(primes-less-than 20)

14. Problema: Scrieți un cod în Racket pentru a verifica dacă o listă de numere este strict crescătoare.

15. Problema: Scrieți un cod în Racket pentru a găsi diferența maximă dintre două numere consecutive dintr-o listă de numere.

```
(define (max-consecutive-difference lst)
  (apply max (map (lambda (x y) (abs (- x y))) lst (cdr lst))))
(max-consecutive-difference '(1 5 2 9 3))
```

Problema: Scrieți un cod în Racket pentru a găsi suma pătratelor numerelor pozitive dintr-o listă de numere.

```
(define (sum-of-positive-squares lst)
  (foldl (lambda (x acc) (if (> x 0) (+ (* x x) acc) acc)) 0 lst))
(sum-of-positive-squares '(-2 3 -1 4 -5))
```

Problema: Scrieți un cod în Racket pentru a găsi toate permutările unei liste de numere.

```
(require math.combinatorics)
(define (permutations lst)
  (list->vector (permutations-of lst)))
(permutations '(1 2 3))
```

Problema: Scrieți un cod în Racket pentru a verifica dacă o listă de numere conține duplicate.

```
(define (has-duplicates? lst)
  (not (equal? (length lst) (length (remove-duplicates lst)))))
(has-duplicates? '(1 2 3 4 5 2))
```

Problema: Scrieți un cod în Racket pentru a verifica dacă toate cuvintele dintr-un text dat sunt unice.

```
(define (all-unique-words? text)
  (let ((words (string-split text)))
    (equal? (length words) (length (remove-duplicates words)))))
(all-unique-words? "apple banana apple pear")
```

Problema: Scrieți un cod în Racket pentru a verifica dacă toate numerele dintr-o listă sunt impare.

```
(define (all-odd-numbers? lst)
  (every odd? lst))
(all-odd-numbers? '(1 3 5 7 9))
```

Problema: Scrieți un cod în Racket pentru a găsi numărul maxim de caractere dintr-o listă de cuvinte.

```
(define (max-characters lst)
  (apply max (map string-length lst)))
(max-characters '("apple" "banana" "pear"))
```

Problema: Scrieți un cod în Racket pentru a găsi suma tuturor numerelor pozitive dintr-o listă de numere.

```
(define (sum-positive-numbers lst)
  (foldl (lambda (x acc) (if (> x 0) (+ x acc) acc)) 0 lst))
(sum-positive-numbers '(-2 3 -1 4 -5))
```

Problema: Scrieți un cod în Racket pentru a găsi toate numerele dintr-o listă care sunt patrate perfecte.

```
(define (perfect-squares lst)
  (filter (lambda (x) (= (sqrt x) (round (sqrt x)))) lst))
(perfect-squares '(1 2 3 4 5 6 7 8 9 10))
```

Problema: Scrieți un cod în Racket pentru a găsi diferența minimă dintre două numere consecutive dintr-o listă de numere.

```
(define (min-consecutive-difference lst)
  (apply min (map (lambda (x y) (abs (- x y))) lst (cdr lst))))
(min-consecutive-difference '(1 5 2 9 3))
```

Problema: Scrieți un cod în Racket pentru a verifica dacă toate cuvintele dintr-o listă sunt anagrame ale unui cuvânt dat.

```
(define (anagrams? word lst)
  (every (lambda (w) (anagram? word w)) lst))
(anagrams? "listen" '("silent" "enlist" "tinsel"))
```

# Semesteri funkcii folog

member( $X, L$ )

append( $L_1, L_2, L_R$ ) sau append([ $L$ ],  $L$ )

prefix ( $P_{\text{last}}, \text{whole}$ )

select( $X, L_1, L_2$ )  $L_1 \setminus X = L_2$  sau. select( $X, X_L, Y, Y_L$ ): select( $B_1, [a, b, c, b],$   
 $[c, d, e, f], X$ ).

nextto( $X, Y, L$ ) ( $X$  &  $Y$  de la  $L$ )

delete( $L, @E, L_2$ )

nth0( $M, L, E, R$ ) (primul element din  $L$  al  $E$ ,  $R = \text{restul}$ )

last, same\_length, reverse, flatten (din [L] fac lista cu toate

subseq, max\_member, min\_member, sum\_list, max\_list, min\_list,  
intersection, union,

pair( $X, Y, [X, Y]$ ).

split( $L, (L_1, L_2)$ ):- neplit( $P_0$ ),  $L_1, L_2, P_0$ .

findall( $P, P, L$ ) lista de fapte care sunt rezultatul  $P$ .

setof( $X, P, L$ ) sa fie o lista de fapte care sunt rezultatul  $P$ .

bagof( $X, P, L$ ) sa fie o lista de fapte care sunt rezultatul  $P$ .

random( $L, H, X$ ), between( $C, H, X$ ), succ( $X, Y$ ), obs( $X$ ), max( $X, Y$ )

rand( $X$ ), trunc( $X$ ), round( $X$ ), ceiling( $X$ ), sqrt( $X$ ).



```

replace([], _, _, []) :- !.
replace([Item1|L1], Item1, Item2, [Item2|R2]) :- replace(L1, Item1, Item2, R2), !.
replace([X|L1], Item1, Item2, [X|R2]) :- replace(L1, Item1, Item2, R2).

reunion([], L2, L2).
reunion(L1, [], L1).
reunion([X|L1], [Y|L2], [X|L3]) :- X < Y, !, reunion(L1, [Y|L2], L3).
reunion([X|L1], [Y|L2], [Y|L3]) :- X > Y, !, reunion([X|L1], L2, L3).
reunion([X|L1], [X|L2], [X|L3]) :- reunion(L1, L2, L3), !.

par(X) :- mod(X, 2) =:= 0.
filter(L1, L2) :- findall(X, (member(X, L1), par(X)), L2).
pare(L1, L2) :- filter(L1, L2).

functie2(X, Acc, [X|Acc]). 
myfold([], Acc, Acc). 
myfold([X|L1], Acc, Y) :- \+ member(X, Acc), !, functie2(X, Acc, Z), myfold(L1, Z, Y). 
myfold([X|L1], Acc, Y) :- member(X, Acc), myfold(L1, Acc, Y). 
set([], []). 
set(L1, L2) :- myfold(L1, [], L2).

functie1(X, Acc, Y) :- Y is X + Acc. 
myfold([], Acc, Acc). 
myfold([X|L1], Acc, Y) :- functie1(X, Acc, Z), myfold(L1, Z, Y).

zipWith([], [], []).
zipWith([X|L1], [Y|L2], [Z|L3]) :- functie1(X, Y, Z), zipWith(L1, L2, L3).

functie(X, Y) :- Y is 2 * X.
mymap([], []).
mymap([X|L1], [Y|L2]) :- functie(X, Y), mymap(L1, L2).

orMap([]) :- fail.
orMap([X|L1]) :- par(X) ; orMap(L1), !.

andMap([]).
andMap([X|L1]) :- par(X), andMap(L1), !.

reverse(X, L) :- reverse(X, L, []).
reverse([], Z, Z).
reverse([H|T], Z, Acc) :- reverse(T, Z, [H|Acc]).

cons(X, L, [X|L]). 
myFoldR(L, Acc, R) :- reverse(L, L1), myFoldL(L1, Acc, R). 
myFoldL([], Acc, Acc). 
myFoldL([X|L1], Acc, Y) :- cons(X, Acc, Z), myFoldL(L1, Z, Y).

adjacent(X, Y, Zs) :- append(_, [X, Y|_], Zs), !.

```

```
last(X, [X], !).
subsequence([X|Xs], [X|Ys]) :- subsequence(Xs, Ys).
subsequence(Xs, [_|Ys]) :- subsequence(Xs, Ys).
subsequence([], _) :- !.
double([], []).
double([X|L1], [X, X|L2]) :- double(L1, L2).
palindrome([]).
palindrome([_]).
palindrome(Pal) :- append([H|T], [H], Pal), palindrome(T), !.

take(1, [H|_], H) :- !.
take(N, [_|T], X) :- N1 is N-1, take(N1, T, X).

zipWith([], [], []).
zipWith([X|L1], [Y|L2], [Z|L3]) :- functie1(X, Y, Z), zipWith(L1, L2, L3).

zip([], [], []).
zip([X|L1], [Y|L2], [(X, Y)|L3]) :- zip(L1, L2, L3).
```

% Exista last(?Elem, ?List) <b>my_last(X,[X]).</b> <b>my_last(X,[_ L]) :- my_last(X,L).</b>	<b>swap([X, Y T], [Y, X T]):-X&gt;Y.</b> <b>swap([Z T], [Z TT]):-swap(T, TT).</b>	<b>busort(L,S):-swap(L,LS),!,busort(LS, S).</b> <b>busort(S,S).</b>
<b>last_but_one(X,[X,_]).</b> <b>last_but_one(X,[_,Y Ys]) :-</b> <b>last_but_one(X,[Y Ys]).</b>	<b>cmmdc(X,0,X) :- X &gt; 0.</b> <b>cmmdc(X,Y,G) :- Y &gt; 0, Z is X mod Y, cmmdc(Y,Z,G).</b>	<b>tree(1,t(a,t(b,t(d,nil,nil),t(e,nil,nil)),t(c,nil,t(f,t(g,nil,nil),nil))))).</b> <b>tree(2,t(a,nil,nil)).</b> <b>tree(3,nil).</b>
<b>element_at(X,[X _],1).</b> <b>element_at(X,[_ L],K) :- K &gt; 1, K1 is K - 1,</b> <b>element_at(X,L,K1).</b>	<b>istree(nil).</b> <b>istree(t(_,_R)) :- istree(L),</b> <b>istree(R).</b>	<b>symmetric(nil).</b> <b>symmetric(t(_,_R)) :- mirror(L,R).</b>
<b>my_length([],0).</b> <b>my_length([_ L],N) :- my_length(L,N1), N is</b> <b>N1 + 1.</b>	<b>min([X], X):-!.</b> <b>min([P R], P):-min(R,X), X &gt; P, !.</b> <b>min([P R],X):-min(R,X), X = = P.</b>	<b>mirror(nil,nil).</b> <b>mirror(t(_,_L1,R1),t(_,_L2,R2)) :-</b> <b>mirror(L1,R2), mirror(R1,L2).</b>
<b>my_reverse(L1,L2) :- my_rev(L1,L2,[]).</b> <b>my_rev([],L2,L2) :- !.</b> <b>my_rev([X Xs],L2,Acc) :-</b> <b>my_rev(Xs,L2,[X Acc]).</b>	<b>split(L,O,[],L).</b> <b>split([X Xs],N,[X Ys],Zs) :- N &gt; 0,</b> <b>N1 is N - 1, split(Xs,N1,Ys,Zs).</b>	<b>remove_at(X,[X Xs],1,Xs).</b> <b>remove_at(X,[Y Xs],K,[Y Ys]) :- K &gt; 1,</b> <b>K1 is K - 1, remove_at(X,Xs,K1,Ys).</b>
<b>is_palindrome(L) :- reverse(L,L).</b>	<b>insert_at(X,L,K,R) :-</b> <b>remove_at(X,R,K,L).</b>	<b>min-sort([], [])):-!.</b> <b>min-sort(L, [M LS]):-min(L, M),</b> <b>select(M, L, LM), min-sort(LM, LS).</b>
<b>my_flatten(X,[X]) :- \+ is_list(X).</b> <b>my_flatten([],[]).</b> <b>my_flatten([X Xs],Zs) :-</b> <b>my_flatten(X,Y), my_flatten(Xs,Ys),</b> <b>append(Y,Ys,Zs).</b>	<b>drop(L1,N,L2) :- drop(L1,N,L2,N).</b> <b>drop([],_,[],_).</b> <b>drop([_ Xs],N,Ys,1) :-</b> <b>drop(Xs,N,Ys,N).</b> <b>drop([X Xs],N,[X Ys],K) :- K &gt; 1, K1</b> <b>is K - 1, drop(Xs,N,Ys,K1).</b>	<b>slice([X _],1,1,[X]).</b> <b>slice([X Xs],1,K,[X Ys]) :- K &gt; 1,</b> <b>K1 is K - 1, slice(Xs,1,K1,Ys).</b> <b>slice([_ Xs],I,K,Ys) :- I &gt; 1,</b> <b>I1 is I - 1, K1 is K - 1,</b> <b>slice(Xs,I1,K1,Ys).</b>
<b>compress([],[]).</b> <b>compress([X],[X]).</b> <b>compress([X,X Xs],Zs) :-</b> <b>compress([X Xs],Zs).</b> <b>compress([X,Y Ys],[X Zs]) :- X \= Y,</b> <b>compress([Y Ys],Zs).</b>	<b>decode([],[]).</b> <b>decode([X Ys],[X Zs]) :- \+</b> <b>is_list(X), decode(Ys,Zs).</b> <b>decode([[1,X] Ys],[X Zs]) :-</b> <b>decode(Ys,Zs).</b> <b>decode([[N,X] Ys],[X Zs]) :- N &gt; 1,</b> <b>N1 is N - 1, decode([[N1,X] Ys],Zs).</b>	<b>count(X,[],[],1,X).</b> <b>count(X,[],[],N,[N,X]) :- N &gt; 1.</b> <b>count(X,[Y Ys],[Y Ys],1,X) :- X \= Y.</b> <b>count(X,[Y Ys],[Y Ys],N,[N,X]) :- N &gt;</b> <b>1, X \= Y.</b> <b>count(X,[X Xs],Ys,K,T) :- K1 is K + 1,</b> <b>count(X,Xs,Ys,K1,T).</b>
<b>transfer(X,[],[],[X]).</b> <b>transfer(X,[Y Ys],[Y Ys],[X]) :- X \= Y.</b> <b>transfer(X,[X Xs],Ys,[X Zs]) :-</b> <b>transfer(X,Xs,Ys,Zs).</b>	<b>dupli([],[]).</b> <b>dupli([X Xs],[X,X Ys]) :-</b> <b>dupli(Xs,Ys).</b>	<b>encode_direct([],[]).</b> <b>encode_direct([X Xs],[Z Zs]) :-</b> <b>count(X,Xs,Ys,1,Z),</b> <b>encode_direct(Ys,Zs).</b>
<b>pack([],[]).</b> <b>pack([X Xs],[Z Zs]) :-</b> <b>transfer(X,Xs,Ys,Z), pack(Ys,Zs).</b>	<b>range(I,I,[I]).</b> <b>range(I,K,[I L]) :- I &lt; K, I1 is I + 1,</b> <b>range(I1,K,L).</b>	<b>has_factor(N,L) :- N mod L =:= 0.</b> <b>has_factor(N,L) :- L * L &lt; N, L2 is L + 2,</b> <b>has_factor(N,L2).</b>
<b>transform([],[]).</b> <b>transform([[X Xs] Ys],[[N,X] Zs]) :-</b> <b>length([X Xs],N), transform(Ys,Zs).</b>	<b>combination(0,_,[]).</b> <b>combination(K,L,[X Xs]) :- K &gt; 0,</b> <b>el(X,L,R), K1 is K-1,</b> <b>combination(K1,R,Xs).</b>	<b>is_prime(2).</b> <b>is_prime(3).</b> <b>is_prime(P) :- integer(P), P &gt; 3, P mod 2 =\= 0, \+ has_factor(P,3).</b>
<b>encode(L1,L2) :- pack(L1,L),</b> <b>transform(L,L2).</b>	<b>el(X,[X L],L).</b> <b>el(X,[_ L],R) :- el(X,L,R).</b>	<b>qsort([],[]).</b> <b>qsort([H T],S) :- split(H,T,L,</b> <b>R),qsort(L,LS), qsort(R,RS),</b> <b>append(LS,[H RS],S).</b>
<b>strip([],[]).</b> <b>strip([[1,X] Ys],[X Zs]) :- strip(Ys,Zs).</b> <b>strip([[N,X] Ys],[[N,X] Zs]) :- N &gt; 1,</b> <b>strip(Ys,Zs).</b>	<b>rotate([],_,[]) :- !.</b> <b>rotate(L1,N,L2) :-</b> <b>length(L1,NL1), N1 is N mod NL1,</b> <b>split(L1,N1,S1,S2),</b> <b>append(S2,S1,L2).</b>	<b>rnd_select(_,0,[]).</b> <b>rnd_select(Xs,N,[X Zs]) :- N &gt; 0,</b> <b>length(Xs,L), I is random(L) + 1,</b> <b>remove_at(X,Xs,I,Ys),</b> <b>N1 is N - 1, rnd_select(Ys,N1,Zs).</b>
<b>encode_modified(L1,L2) :-</b> <b>encode(L1,L), strip(L,L2).</b>	<b>lotto(N,M,L) :- range(1,M,R),</b> <b>rnd_select(R,N,L).</b>	<b>rnd_permu(L1,L2) :- length(L1,N),</b> <b>rnd_select(L1,N,L2).</b>

1. `member/2`: Verifică dacă un element aparține unei liste.

```
member(X, [X|_]).  
member(X, [_|T]) :- member(X, T).
```

2. `append/3`: Concatenează două liste.

```
append([], List, List).  
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

3. `length/2`: Calculează lungimea unei liste.

```
length([], 0).  
length([_|T], N) :- length(T, N1), N is N1 + 1.
```

4. `reverse/2`: Inversează ordinea elementelor unei liste.

```
reverse(List, Reversed) :- reverseAcc(List, [], Reversed).  
reverseAcc([], Acc, Acc).  
reverseAcc([H|T], Acc, Reversed) :- reverseAcc(T, [H|Acc], Reversed).
```

5. `sum\_list/2`: Calculează suma elementelor unei liste.

```
sum_list([], 0).  
sum_list([X|T], Sum) :- sum_list(T, Rest), Sum is X + Rest.
```

6. `max\_list/2`: Găsește valoarea maximă dintr-o listă.

```
max_list([X], X).  
max_list([H|T], Max) :- max_list(T, RestMax), Max is max(H, RestMax).
```

7. `min\_list/2`: Găsește valoarea minimă dintr-o listă.

```
min_list([X], X).  
min_list([H|T], Min) :- min_list(T, RestMin), Min is min(H, RestMin).
```

8. `sort/2`: Sortează o listă în ordine crescătoare.

```
sort(List, Sorted) :- predsort(@=<, List, Sorted).
```

9. `delete/3`: Elimină prima apariție a unui element dintr-o listă.

```
delete(X, [X|T], T).  
delete(X, [H|T], [H|Result]) :- delete(X, T, Result).
```

10. `select/3`: Elimină prima apariție a unui element dintr-o listă și întoarce restul listei.

```
select(X, [X|T], T).  
select(X, [H|T], [H|Result]) :- select(X, T, Result).
```

11. `nth0/3`: Accesează al N-lea element dintr-o listă (indexare de la 0).

```
nth0(0, [X|_], X).  
nth0(N, [_|T], X) :- N > 0, N1 is N - 1, nth0(N1, T, X).
```

12. `nth1/3`: Accesează al N-lea element dintr-o listă (indexare de la 1).

```
nth1(1, [X|_], X).  
nth1(N, [_|T], X) :- N > 1, N1 is N - 1, nth1(N1, T, X).
```

13. `sublist/2`: Verifică dacă o listă este sublista unei alte liste.

```
sublist(Sublist, List) :- append(_, Rest, List), append(Sublist, _, Rest).
```

14. `subset/2`: Verifică dacă o listă este un submulțime a unei alte liste.

```
subset([], _).  
subset([X|T], Set) :- member(X, Set), subset(T, Set).
```

15. `flatten/2`: Transformă o listă multidimensională într-o listă unidimensională.

```
flatten(List, Flattened) :- flattenAcc(List, [], Flattened).  
flattenAcc([], Acc, Acc).  
flattenAcc([H|T], Acc, Flattened) :- flattenAcc(T, Acc1, Flattened1), append(H, Flattened1, Acc1).
```

16. `zip/3`: Creează o listă de perechi folosind elemente din două liste.

```
zip([], [], []).  
zip([X|T1], [Y|T2], [(X,Y)|Zipped]) :- zip(T1, T2, Zipped).
```

17. `maplist/2`: Aplică un predicat asupra fiecărui element al unei liste.

```
maplist(_, []).  
maplist(Pred, [X|Xs]) :- call(Pred, X), maplist(Pred, Xs).
```

18. `filter/3`: Filtrează elementele unei liste pe baza unui predicat.

```
filter(_, [], []).  
filter(Pred, [X|Xs], Filtered) :- (call(Pred, X) -> Filtered = [X|Rest] ; Filtered = Rest), filter(Pred, Xs, Rest).
```

19. `partition/4`: Separă elementele unei liste în două liste, pe baza unui predicat.

```
partition(_, [], [], []).  
partition(Pred, [X|Xs], [X|True], False) :- call(Pred, X),  
partition(Pred, Xs, True, False).  
partition(Pred, [X|Xs], True, [X|False]) :- \+ call(Pred, X),  
partition(Pred, Xs, True, False).
```

20. `findall/3`: Colecțează toate soluțiile unui predicat într-o listă.

```
findall(X, Goal, List) :- bagof(X, Goal, List), !.  
findall(_, _, []).
```

21. `forall/2`: Verifică dacă un predicat este adevărat pentru toate soluțiile date de un alt predicat.

```
forall(Condition, Action) :- \+ (Condition, \+ Action).
```

22. `once/1`: Caută o singură soluție a unui predicat, ignorând alte posibile soluții.

```
once(Goal) :- Goal, !.
```

23. `between/3`: Generează numere întregi într-un interval specificat.

```
between(Min, Max, N) :- Min <= Max, (Min = N ; (Min1 is Min+1,  
between(Min1, Max, N))).
```

24. `atom/1`: Verifică dacă un termen este un atom.

```
atom(X)  
:- atomic(X).
```

25. `number/1`: Verifică dacă un termen este un număr.

```
number(X) :- atomic(X), atom_number(X, _).
```

26. `atomic\_list\_concat/2`: Concatenează o listă de atomi într-un atom.

```
atomic_list_concat(List, Atom) :- atomic_list_concat(List, '',  
Atom).
```

27. `atomic\_concat/3`: Concatenează două atomi într-un alt atom.

```
atomic_concat(Atom1, Atom2, Concat) :- atom_concat(Atom1, Atom2,  
Concat).
```

28. `atomic\_length/2`: Calculează lungimea unui atom.

```
atomic_length(Atom, Length) :- atom_length(Atom, Length).
```

29. `call/1`: Execută un termen ca un apel la un predicat.  
call(Term) :- call(Term).

30. `ground/1`: Verifică dacă un termen este instantiat în întregime.  
ground(X) :- \+ var(X), X \= [], X \= [\_|\_], X \= (\_,\_).

1. Predicatul `even\_list(+List)` care verifică dacă toate elementele unei liste sunt numere pare:

even\_list([]).

even\_list([H|T]) :- 0 is H mod 2, even\_list(T).

2. Predicatul `merge\_lists(+List1, +List2, -Merged)` care combină două liste într-o singură, prin intercalarea elementelor:

merge\_lists([], List, List).

merge\_lists(List, [], List).

merge\_lists([H1|T1], [H2|T2], [H1, H2|Merged]) :- merge\_lists(T1, T2, Merged).

3. Predicatul `palindrome(+List)` care verifică dacă o listă este palindrom (se citește la fel în ambele sensuri):

palindrome(List) :- reverse(List, List).

4. Predicatul `remove\_duplicates(+List, -WithoutDuplicates)` care elimină duplați dintr-o listă:

remove\_duplicates([], []).

remove\_duplicates([H|T], WithoutDuplicates) :- member(H, T), !,  
remove\_duplicates(T, WithoutDuplicates).

remove\_duplicates([H|T], [H|WithoutDuplicates]) :-  
remove\_duplicates(T, WithoutDuplicates).

5. Predicatul `sublist(+Sublist, +List)` care verifică dacă `Sublist` este o sublistă a lui `List`:

sublist(Sublist, List) :- append(\_, Sublist, Suffix), append(Suffix, \_, List).

6. Predicatul `sum\_list(+List, -Sum)` care calculează suma elementelor unei liste:

sum\_list([], 0).

sum\_list([H|T], Sum) :- sum\_list(T, Rest), Sum is H + Rest.

7. Predicatul `last\_element(+List, -Last)` care găsește ultimul element al unei liste:

last\_element([X], X).

last\_element([\_|T], Last) :- last\_element(T, Last).

8. Predicatul `even\_odd\_list(+List, -Even, -Odd)` care separă elementele pare și impare dintr-o listă în două liste separate:

```
even_odd_list([], [], []).
```

```
even_odd_list([H|T], [H|Even], Odd) :- 0 is H mod 2, even_odd_list(T, Even, Odd).
```

```
even_odd_list([H|T], Even, [H|Odd]) :- 1 is H mod 2, even_odd_list(T, Even, Odd).
```

9. Predicatul `flatten\_list(+List, -Flattened)` care aplanează o listă multidimensională într-o listă unidimensională:

```
flatten_list([], []).
```

```
flatten_list([H|T], Flattened) :- is_list(H), flatten_list(H, FlatH), flatten_list(T, FlatT), append(FlatH, FlatT, Flattened).
```

```
flatten_list([H|T], [H|Flattened]) :- \+ is_list(H), flatten_list(T, Flattened).
```

10. Predicatul `rotate\_list(+List, +N, -Rotated)` care rotește o listă la dreapta de N ori:

```
rotate_list(List, 0, List).
```

```
rotate_list([H|T], N, Rotated) :- N > 0, append(T, [H], NewList), N1 is N - 1, rotate_list(NewList, N1, Rotated).
```

11. Predicatul `max\_list(+List, -Max)` care găsește valoarea maximă dintr-o listă:

```
max_list([X], X).
```

```
max_list([H|T], Max) :- max_list(T, RestMax), Max is max(H, RestMax).
```

12. Predicatul `delete\_element(+Element, +List, -Result)` care șterge toate aparițiile unui element dintr-o listă:

```
delete_element(_, [], []).
```

```
delete_element(X, [X|T], Result) :- delete_element(X, T, Result).
```

```
delete_element(X, [H|T], [H|Result]) :- X \= H, delete_element(X, T, Result).
```

13. Predicatul `split\_list(+List, +N, -Prefix, -Suffix)` care împarte o listă în două părți: prefixul de lungime N și sufixul:

```
split_list(List, N, Prefix, Suffix) :- length(Prefix, N), append(Prefix, Suffix, List).
```

14. Predicatul `is\_sorted(+List)` care verifică dacă o listă este sortată în ordine crescătoare:

```
is_sorted([]).
```

```
is_sorted([_]).
```

```
is_sorted([X, Y|T]) :- X == Y, is_sorted([Y|T]).
```

15. Predicatul `intersection\_list(+List1, +List2, -Intersection)` care găsește intersecția a două liste (elementele comune):

```
intersection_list([], _, []).
intersection_list([H|T], List2, [H|Intersection]) :- member(H, List2), intersection_list(T, List2, Intersection).
intersection_list([_|T], List2, Intersection) :- intersection_list(T, List2, Intersection).
```

16. Predicatul `alternating\_list(+List, -Alternating)` care construiește o listă formată din elementele alternative dintr-o listă:

```
alternating_list([], []).
alternating_list([X], [X]).
alternating_list([X, _|T], [X|Alternating]) :- alternating_list(T, Alternating).
```

17. Predicatul `duplicate\_list(+List, -Duplicated)` care dublează fiecare element dintr-o listă:

```
duplicate_list([], []).
duplicate_list([X|T], [X, X|Duplicated]) :- duplicate_list(T, Duplicated).
```

18. Predicatul `merge\_sort(+List, -Sorted)` care sortează o listă folosind algoritmul Merge Sort:

```
merge_sort([], []).
merge_sort([X], [X]).
merge_sort(List, Sorted) :- divide_list(List, Left, Right),
merge_sort(Left, SortedLeft), merge_sort(Right, SortedRight),
merge_lists(SortedLeft, SortedRight, Sorted).
divide_list(List, Left, Right) :- length(List, Len), HalfLen is Len // 2,
length(Left, HalfLen), append(Left, Right, List).
merge_lists([], List, List).
merge_lists(List, [], List).
merge_lists([X|T1], [Y|T2], [X|Sorted]) :- X <= Y, merge_lists(T1, [Y|T2], Sorted).
merge_lists([X|T1], [Y|T2], [Y|Sorted]) :- X > Y, merge_lists([X|T1], T2, Sorted).
```

19. Predicatul `insert\_sorted\_list(+Element, +List, -Sorted)` care inserează un element într-o listă sortată, menținând ordinea:

```
insert_sorted_list(X, [], [X]).
insert_sorted_list(X, [H|T], [X, H|T]) :- X <= H.
insert_sorted_list(X, [H|T], [H|Sorted]) :- X > H,
insert_sorted_list(X, T, Sorted).
```

20. Predicatul `remove\_nth(+N, +List, -Result)` care elimină al N-lea element dintr-o listă:

```
remove_nth(_, [], []).  
remove_nth(1, [_|T], T).  
remove_nth(N, [H|T], [H|Result]) :- N > 1, N1 is N - 1,  
remove_nth(N1, T, Result).
```



2. Care dintre cele două adunări se vor realiza în codul Racket de mai jos? Justificați!

```
(define y 10)
((lambda (x) (lambda (y) (if (> x 2) (+ 1 y) (+ x y))) ) 5)
```

*Soluție:*

Niciuna, pentru că  $\lambda y.$  nu este aplicat. Expresia întoarce rezultatul aplicării lui  $\lambda x.$ , care este o procedură.

3. Date fiind două liste de numere L1 și L2, scrieți în Racket codul care produce o listă de perechi de forma  $(x . L)$ , unde x este un element din L1, iar L este lista elementelor din L2 care sunt divizori ai lui x. E.g. pentru  $L1 = (25 30 100)$  și  $L2 = (2 3 5)$  rezultatul este  $((25 . (2 5)) (30 . (2 3 5)) (100 . (2 5)))$ . Nu folosiți recursivitate explicită. Explicați cum funcționează codul.

*Soluție:*

```
(define (divpairs L1 L2) (map
  (λ (x) (cons x (filter (λ (n) (zero? (remainder x n))) L2))) L1))
(divpairs '(25 30 100) '(2 3 5))
```

4. a. Ce efect are următorul cod Racket:

```
(define a (lambda (f g L) (if (null? L) '()
  (append (if (f (car L)) (list (g (car L))) '()) (a f g (cdr L)))))))
```

- b. Rescrieți funcția cu funcționale, evitând recursivitatea explicită.

*Soluție:*

a. Aplică funcția g fiecărui element din L pentru care funcția f este adeverată și întoarce o listă cu rezultatele acestor aplicări.

b. `(define (afunc f g L) (map g (filter f L)))`

2. Care dintre cele două adunări se vor realiza în codul Racket de mai jos? Justificați!

```
(define y 10)
((lambda (x) (if (> x 2) (lambda (y) (+ y 1)) (+ x y))) 5)
```

*Soluție:*

Niciuna, pentru că suntem pe ramura de true a if-ului, și apoi  $\lambda y.$  nu se aplică. Rezultatul expresiei este o procedură.

3. Date fiind două liste de numere L1 și L2, scrieți în Racket codul care produce o listă de perechi de forma  $(x . L)$ , unde x este un element din L1, iar L este lista elementelor din L2 care sunt multipli ai lui x. E.g. pentru  $L1 = (2 3 4)$  și  $L2 = (4 5 8 9 10 100)$  rezultatul este  $((2 . (4 8 10 100)) (3 . (9)) (4 . (4 8 100)))$ . Nu folosiți recursivitate explicită. Explicați cum funcționează codul.

*Soluție:*

```
(define (mulpairs L1 L2) (map
  (λ (x) (cons x (filter (λ (n) (zero? (remainder n x))) L2))) L1))
(mulpairs '(2 3 4) '(4 5 8 9 10 100))
```

4. a. Ce efect are următorul cod Racket:

```
(define b (lambda (f g L) (if (null? L) '()
    (append (if (f (g (car L))) (list (g (car L))) '()) (b f g (cdr L))))))
```

b. Rescrieți funcția cu funcționale, evitând recursivitatea explicită.

*Soluție:*

a. Aplică funcția g pe fiecare element din L, apoi păstrează în rezultat doar acele rezultate pentru care funcția f este adevărată.

```
b. (define (bfunc f g L) (filter f (map g L)))
```

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))))
```

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 și/sau liste de numere (maxim 2 niveluri de imbricare). Să se definească, în Racket, funcția `count-nulls` care numără listele vide aflate pe orice nivel de imbricare.

ex: (`count-nulls '(() 1 (2 () (3 4 5)) ()))`) va întoarce 3, pentru că avem o listă vidă direct în L, și încă 2 în liste interioare lui L (acesta fiind și nivelul maxim de imbricare admis, deci nu trebuie căutate liste vide la adâncime mai mare, de exemplu `'(0 (1 (2 ()))`) nu este un input valid pentru problemă).

*Soluție:*

```
(define (count-nulls L)
  (let* ((surface-nulls (length (filter null? L)))
        (inner-lists (filter list? L))
        (all-inner (apply append inner-lists))
        (deep-nulls (length (filter null? all-inner))))
    (+ surface-nulls deep-nulls)))
```

4. Definiți în Racket următorul flux:

```
'((1) (2 1 2) (3 2 1 2 3) (4 3 2 1 2 3 4) (5 4 3 2 1 2 3 4 5) ...)
```

Soluție:

```
(define my-stream
  (stream-cons '(1)
    (stream-map (lambda (L) (let ([new (list (add1 (car L)))]
                                    (append new L new)))
                  my-stream)))
```

3. **Atenție:** Acest exercițiu nu se punctează decât dacă este rezolvat fără a folosi recursitate 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 `longest-list` care întoarce lista de lungime maximă din input - fie întreg  $L$ , fie una din listele interioare lui  $L$ .

ex: `(longest-list '(0 1 2 (3 4 3 4 3 4 3 4 3 4) 5 (6) 7))` va întoarce `'(3 4 3 4 3 4 3 4 3 4)` - pt ca are lungimea 10

`(longest-list '(0 1 2 (3 4 3 4 3 4) 5 (6) 7))` va întoarce `'(0 1 2 (3 4 3 4 3 4) 5 (6) 7)` - pt ca are lungimea 7

Soluție:

```
(define (longest-list L)
  (let ((inner-lists (filter list? L)))
    (foldr (lambda (L acc) (if (> (length L) (length acc)) L acc))
           L
           inner-lists)))
```

4. Definiți în Racket fluxul coeficienților binomiali:

```
'((1) (1 1) (1 2 1) (1 3 3 1) (1 4 6 4 1) (1 5 10 10 5 1) ...)
```

Soluție:

```
(define binomial-stream
  (stream-cons '(1)
    (stream-map (lambda (L) (let ([M (cons 0 L)])
                               (map + M (reverse M))))
               binomial-stream)))
```

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 de numere. Să se definească, în Racket, funcția `count-smaller` care întoarce o listă de perechi de forma (`număr_n_din_L . câte_valori_din_L_sunt_mai_mici_ca_n`). Atenție, fiecare număr din L trebuie să apară o singură dată în partea din stânga a perechilor!

ex: `(count-smaller '(1 2 3 2 4 3 1 2))` va întoarce `'((4 . 7) (3 . 5) (2 . 2) (1 . 0))` - sau aceleași 4 perechi în orice altă ordine (ordinea NU contează), pentru că în L sunt 7 numere mai mici ca 4, 5 mai mici ca 3, 2 mai mici ca 2, și niciunul mai mic ca 1.

*Soluție:*

```
(define (count-smaller L)
  (let ((no-dups (foldl (lambda (n acc) (if (member n acc) acc (cons n
    → acc))) '() L)))
    (map (lambda(n) (cons n (length (filter (lambda (x) (< x n)) L))))
      → no-dups)))
```

4. Se dau în Racket două fluxuri de liste, `s1` și `s2`. Definiți fluxul ale căror elemente se obțin prin intersectarea listelor de pe aceeași poziție din `s1` și `s2`.

Exemplu: `s1 = '((1 2) (3 4 5) ...)`, `s2 = '((2 3) (1 4 5) ...)`, `rezultat = '((2) (4 5) ...)`

*Soluție:*

```
(define result
  (stream-zip-with (lambda (L1 L2) (filter (lambda (e) (member e L1)) L2))
    → s1 s2))
```

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))
```

2. Se dă următorul cod Racket:

```
(define computation (λ () (equal? 5 5)))  
(define (f x) (and (> x 5) (computation)))  
(filter f '(1 3 5 7 9))
```

- (a) De câte ori se apelează funcția `equal?` ?  
(b) Rescrieți codul pentru `computation` și pentru `f` folosind promisiuni (pentru întârzierea lui `computation`) și răspundeți din nou la întrebarea (a).

*Soluție:*

- (a) de 2 ori (pentru fiecare element mai mare decât 5)  
(b) 

```
(define computation (delay (equal? 5 5)))  
(define (f x) (and (> x 5) (force computation)))  
(filter f '(1 3 5 7 9))
```

acum se apelează o singură dată, la prima evaluare a lui `computation`.

3. Dată fiind o listă de liste de numere LL, scrieți în Racket codul care produce sublista lui LL în care pentru toate elementele L suma elementelor este cel puțin egală cu produsul lor. E.g. pentru L = ((1 2 3) (1 2) (4 5) (.5 .5)) rezultatul este ((1 2 3) (1 2) (0.5 0.5)). Nu folosiți recursivitate explicită.

*Soluție:*

```
(filter (lambda (L) (>= (apply + L) (apply * L))) '((1 2 3) (1 2) (4 5) (.5 .5)))
```

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.

5. (a) Câte aplicații ale funcției de incrementare sunt calculate pentru evaluarea expresiei Racket  
`(length (map add1 '(1 2 3 4 5 6 7 8 9 10)))` ?

- (b) Dar pentru expresia Haskell `length $ map (+ 1) [1 .. 10]` ?

*Soluție:*

- (a) Toate elementele listei sunt evaluate, deci 10.

- (b) Elementele listei nu sunt evaluate, deci 0.

2. Care este diferența între următoarele două linii de cod Racket

```
(let ((a 1) (b 2)) (let ((b 3) (c (+ b 2))) (+ a b c)))  
(let* ((a 1) (b 2)) (let* ((b 3) (c (+ b 2))) (+ a b c)))
```

*Soluție:*

În prima definiția (b 3) nu este vizibilă în legarea lui c; rezultatul este 8, iar în a doua linie rezultatul este 9.

3. Scrieți în Racket o funcție echivalentă cu `zip` din Haskell, știind că `zip :: [a] -> [b] -> [(a, b)]`. Folosiți cel puțin o funcțională.

*Soluție:*

```
(define (zip L1 L2) (map cons L1 L2))
```

2. Care este diferența între următoarele două linii de cod Racket

```
(let* ((a 1) (b 2) (c (+ a 2))) (+ a b c))  
((lambda (a b c) (+ a b c)) 1 2 (+ a 2))
```

*Soluție:*

În a doua linie a nu este vizibil la invocarea funcției  $\lambda$ ; prima linie dă 6, a doua dă eroare.

3. Scrieți în Racket o funcție echivalentă cu `unzip` din Haskell, știind că `unzip :: [(a, b)] -> ([a], [b])`. Folosiți cel puțin o funcțională.

*Soluție:*

```
(define (unzip L) (cons (map car L) (map cdr L)))
```

2. Este vreo diferență (ca efect, la execuție) între cele două linii de cod Racket? Dacă da, care este diferența?; dacă nu, de ce nu diferă?

```
(let ((a 1)) (let ((b a)) (+ a b)))  
(let* ((a 1) (b a)) (+ a b))
```

*Soluție:*

Nu este nicio diferență; `let*` este același lucru cu câte un `let` imbricat pentru fiecare definiție.

3. Implementați în Racket funcția `f` care primește o listă și determină cel mai mare element. Folosiți, în mod obligatoriu, cel puțin o funcțională.

*Soluție:*

```
(car (filter (λ(e) (null? (filter ((curry <) e) L))) L))  
sau  
(car (filter (λ(e) (null? (filter (λ(a) (< e a)) L))) L))  
sau  
(last (sort L <)))
```

2. Este vreo diferență (ca efect, la execuție) între cele două linii de cod Racket? Dacă da, care este diferența?; dacă nu, de ce nu diferă?

```
(define a 2) (let ((a 1) (b a)) (+ a b))
(define a 2) (letrec ((a 1) (b a)) (+ a b))
```

*Soluție:*

În prima linie, definiția `(a 1)` este vizibilă în corpul `let`-ului, dar nu și în definiția lui `b`, care vede încă `a=2`; prima linie dă 3, a două dă 2.

3. Implementați în Racket funcția `f` care primește o listă și determină cel mai mic element. Folosiți, în mod obligatoriu, cel puțin o funcțională.

*Soluție:*

```
(car (filter (λ(e) (null? (filter ((curry >) e) L))) L))
sau
(car (filter (λ(e) (null? (filter (λ(a) (> e a)) L))) L))
sau
(last (sort L >)))
```

2. Este vreo diferență (ca efect, la execuție) între cele două linii de cod Racket? Dacă da, care este diferența?; dacă nu, de ce nu diferă?

```
(define a 2) (let ((c 2)) (let ((a 1) (b a)) (+ a b)))
(define a 2) (let* ((c 2) (a 1) (b a)) (+ a b))
```

*Soluție:*

În prima linie, definiția `(a 1)` este vizibilă în corpul `let`-ului, dar nu și în definiția lui `b`, care vede încă `a=2`; prima linie dă 3, a două dă 2.

3. Implementați în Racket funcția `f` care primește o listă și determină elementul cu cel mai mare modul. Folosiți, în mod obligatoriu, cel puțin o funcțională.

*Soluție:*

```
(car (filter (λ(e) (null? (filter (compose ((curry <) (abs e)) abs) L))) L))
sau
(car (filter (λ(e) (null? (filter (λ(a) (< (abs e) (abs a))) L))) L))
sau
(let ((M (last (sort (map abs L) <)))) (if (member M L) M (- 0 M))))
```

2. Este vreo diferență (ca efect, la execuție) între cele două linii de cod Racket? Dacă da, care este diferența?; dacă nu, de ce nu diferă?

```
(let ((a 1) (b 2)) (+ a b))
((lambda (a b) (+ a b)) 1 2)
```

*Soluție:*

Nu este nicio diferență; `(let ((ai vi)) corp)` este echivalent cu `(lambda (ai) corp)` aplicat parametrilor  $v_i$

3. Implementați în Racket funcția `f` care primește o listă și determină elementul mai mare decât modulul oricărui alt element. Folosiți, în mod obligatoriu, cel puțin o funcțională.

*Soluție:*

```
(car (filter (λ(e) (null? e) (filter (compose ((curry <) e) abs) L))) L))  
sau  
(car (filter (λ(e) (null? e) (filter (λ(a) (< e (abs a))) L))) L))  
sau  
(last (sort L <))
```

2. Implementați în Racket o funcție `myAndMap` care să aibă un comportament similar cu `andmap` – primește o listă și întoarce o valoare booleană egală cu rezultatul operației `and` pe elementele listei. Folosiți cel puțin o funcțională. Nu folosiți `andmap`.

*Soluție:*

```
(define (myAndMap L) (foldl (λ(x y) (and x y)) #t L)) (am acceptat și foldl/r direct cu and, soluție cu filter, etc)
```

3. Ce întoarce următoarea expresie în Racket? Justificați!

```
(let ((n 2))  
  (letrec ((f (lambda (n)  
              (if (zero? n) 1 (* n (f (- n 1)))))))  
    (f 5)))  
)
```

*Soluție:*

Este factorial.  $5! = 120$ . `n` din let nu are niciun efect pentru că în cod se folosește `n` legat de `lambda`.

4. Cum se poate îmbunătăți următorul cod Racket pentru ca funcția `calcul-complex` să se evalueze doar atunci când este necesar, adică doar atunci când `variant` este fals (fără a muta apelul lui `calcul-complex` în interiorul lui `calcul`) ?

1. `(define (calcul x y z) (if x y z))`
2. `(define (test variant) (calcul variant 2 (calcul-complex 3)))`

*Soluție:*

1. `(define (calcul x y z) (if x y (force z)))`
2. `(define (test variant) (calcul variant 2 (delay (calcul-complex 3))))`

Se poate și folosind închidere lambda și `(z)`, `if` peste apelul lui `calcul-complex`, sau chiar `quote` și `eval`.

2. Implementați în Racket o funcție `myOrMap` care să aibă un comportament similar cu `ormap` – primește o listă și întoarce o valoare booleană egală cu rezultatul operației `or` pe elementele listei. Folosiți cel puțin o funcțională. Nu folosiți `ormap`.

*Soluție:*

```
(define (myOrMap L) (foldl (λ (x y) (or x y)) #f L)) (am acceptat și foldl/r direct cu or, soluție cu filter, etc)
```

3. Ce întoarce următoarea expresie în Racket? Justificați!

```
(letrec ((f (lambda (n)
  (let ((n (- n 1)))
    (if (eq? n -1) 1 (* (+ n 1) (f n)))))))
  (f 5))
)
```

*Soluție:*

Este factorial.  $5! = 120$ .

4. Cum se poate îmbunătăți următorul cod Racket pentru ca funcția `calcul-complex` să se evalueze doar atunci când este necesar, adică doar atunci când `variant` este fals (fără a o muta apelul lui `calcul-complex` în interiorul lui `calcul`) ?

1. `(define (calcul x y z) (if x y z))`
2. `(define (test variant) (calcul variant 2 (calcul-complex 3)))`

*Soluție:*

1. `(define (calcul x y z) (if x y (force z)))`
2. `(define (test variant) (calcul variant 2 (delay (calcul-complex 3))))`

Se mai poate și folosind închidere lambda și `(z)`, if peste apelul lui `calcul-complex`, sau chiar `quote` și `eval`.

2. Scrieți o funcție `setN` în Racket care primește două liste `L1` și `L2` (fără duplicate) ca argumente și întoarce o listă care este intersecția celor două liste, luate ca multimi (rezultatul nu trebuie să conțină duplicate).

*Soluție:*

```
(define (setU L1 L2)
  (cond
    ((null? L1) L2)
    ((member (car L1) L2) (setU (cdr L1) L2))
    (else (cons (car L1) (setU (cdr L1) L2))))
  )) sau
(define (setU2 L1 L2) (foldr (λ (x L)
  (if (member x L) L (cons x L))) '() (append L1 L2)))
```

3. Date fiind funcțiile E și F și următorul cod care considerăm că se execută fără erori, de câte ori sunt evaluate fiecare dintre cele două funcții, și la ce linii din cod se fac evaluările?

```
1. (define fmic (λ (a)
2.   (let [ (f (F a)) (g (delay (E a))) ]
3.     f )  ))
4. (fmic (E 'argument))
```

*Soluție:*

E la 4, F la 2

2. Scrieți o funcție setU în Racket care primește două liste L1 și L2 (fără duplicate) ca argumente și întoarce o listă care este reuniunea celor două liste, luate ca mulțimi (rezultatul nu trebuie să conțină duplicate).

*Soluție:*

```
(define (setN L1 L2)
  (cond
    ((null? L1) '())
    ((member (car L1) L2) (cons (car L1) (setN (cdr L1) L2)))
    (else (setN (cdr L1) L2)))
  )) sau
(define (setN2 L1 L2) (filter (λ (x) (member x L2)) L1))
```

3. Date fiind funcțiile E și F și următorul cod care considerăm că se execută fără erori, de câte ori sunt evaluate fiecare dintre cele două funcții, și la ce linii din cod se fac evaluările?

```
1. (define gmic (λ (a)
2.   (let [ (f (delay (F a))) (x (g a)) ]
3.     f )  ))
4. (gmic (E 'argument))
```

*Soluție:*

E la 4, F niciodată.

2. Implementați o funcție în Racket care ia o listă de numere L1 ca prim parametru și o listă de liste L2 ca al doilea parametru și întoarce acele liste din L2 ale căror lungimi se regăsesc în L1. Utilizați funcționale și nu utilizați recursivitate explicită – soluțiile care nu respectă cele două constrângeri nu vor fi punctate.

Exemplu: (f '(1 2 3 4) '((4 5 6) () (a b))) → '((4 5 6) (a b))

*Soluție:*

```
(λ (L1 L2) (filter (λ (l) (member (length l) L1)) L2))
```

3. La ce se evaluează următoarea expresie în Racket? (cu  $() \equiv []$ )

```
(let* [(x 1) (y 2) (f (delay (λ (y) (+ x y))))] (let [(x 5)] ((force f) x)))
```

*Soluție:*

6.  $x=5 + x=1$ .

2. La ce se evaluează următoarea expresie în Racket? (cu  $() \equiv []$ )

```
(let* [(x 3) (y 4) (f (delay (λ (y) (+ x y))))] (let [(x 1)] ((force f) x)))
```

*Soluție:*

4.  $x=1 + x=3$ .

3. Implementați o funcție în Racket care ia o listă de numere L1 ca prim parametru și o listă de liste L2 ca al doilea parametru și întoarce acele liste din L2 ale căror lungimi nu se regăsesc în L1. Utilizați funcționale și nu utilizați recursivitate explicită – soluțiile care nu respectă cele două constrângeri nu vor fi punctate.

Exemplu:  $(f '(1 0 4 5) '((4 5 6) () (a b))) \rightarrow '((4 5 6) (a b))$

*Soluție:*

```
(λ (L1 L2) (filter (λ (l) (not (member (length l) L1))) L2))
```

2. Implementați funcția `(oddStarts L)` în Scheme, care pentru o listă de liste întoarce câte dintre listele componente încep cu un număr impar. Utilizați **funcționale** și nu utilizați recursivitate explicită – soluțiile care nu respectă cele două constrângeri **nu** sunt punctate.

Exemplu:  $(oddStarts '((2 3 4) (1 2 3) (5 6) (8 9))) \rightarrow 2$

*Soluție:*

```
(define (oddStarts L) (length (filter odd? (map car L))))
```

3. Ce întoarce următoarea expresie în Scheme? Justificați!

```
(apply
```

```
  (lambda (x y) (let ([x 1] [y 2]) (+ x y)))
```

```
  (let ([x 2] [y 3]) (list x y)))
```

```
)
```

4. Ce afișează următorul cod:

1. `(define f (delay (lambda (a) (display a) (newline))))`
2. `(define ff (force f))`
3. `(and (ff 1) (ff 2) #f)`

*Soluție:*

```
1  
2  
#f
```





6. Instantiați clasa `Eq` pentru funcții Haskell care iau un argument numeric, astfel încât două funcții sunt “egale” dacă valoarea lor este egală pentru fiecare număr întreg între 1 și 10.
7. Construiți în Haskell funcția `perms :: [Char] -> [[Char]]` care primește o listă de caractere și întoarce fluxul sirurilor formate din aceste caractere, începând cu siruri de lungime 1. De exemplu: `take 45 $ perms "abc" -> ["a","b","c","aa","ba","ca", ... , "bc","cc","aaa","baa","caa","aba","bba", ... , "acc","bcc","ccc","aaaa","baaa","caaa","abaa","bbaa","cbaa"]`

Construiesc sirurile de lungime 1, apoi celelalte siruri sunt siruri deja în flux la care adaug unul dintre caracterele din alfabetul dat.

7. Implementați în Racket fluxul în care primele 3 elemente sunt 1, 2 și 3, iar fiecare dintre următoarele elemente este produsul dintre cele trei elemente anterioare.

*Soluție:*

```
(define (stream-zip3 f s1 s2 s3) (stream-cons (f (stream-first s1) (stream-first s2) (stream-first s3))
(stream-zip3 f (stream-rest s1) (stream-rest s2) (stream-rest s3)) ))
(define superProducts (stream-cons 1 (stream-cons 2 (stream-cons 3
(stream-zip3 * superProducts
(stream-rest superProducts) (stream-rest (stream-rest superProducts)) ) )))) (stream->list (stream-take
superProducts 7))
```

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)
```

7. Implementați în Haskell, fără a utiliza recursivitate explicită, funcția `setD` care realizează diferența a două mulțimi a și b ( $a \setminus b$ ) date ca liste (fără duplicate). Care este tipul funcției?

*Soluție:*

```
setD a b = [x | x <- a, not $ elem x b]
sau
setD a b = filter (not . (flip elem) b) a
setD :: Eq t => [t] -> [t] -> [t]
```

7. Implementați în Haskell, fără a utiliza recursivitate explicită, funcția `setN` care realizează intersecția a două mulțimi a și b date ca liste (fără duplicate). Care este tipul funcției?

*Soluție:*

```
setN a b = [x | x <- a, elem x b]
sau
setN a b = filter ((flip elem) b) a
setN :: Eq t => [t] -> [t] -> [t]
```

6. Scrieți o funcție Haskell care elimină dintr-o listă elementele care apar de mai multe ori.  
E.g. [1, 2, 3, 2, 3, 5] → [1, 5].

*Soluție:*

```
nodups [] = []
nodups l = head l : nodups (filter (not . (== head l)) $ tail l)
```

6. Scrieți o funcție Haskell care păstrează dintr-o listă doar valorile care apar de mai multe ori. E.g. [1, 2, 3, 2, 3] → [2, 3].

*Soluție:*

```
dups [] = []
dups (h:t)
| elem h t = h : dups (filter (/= h) t)
| otherwise = dups t
```

7. Care este fluxul s pentru care este adevărat:

```
(take 10 $ zipWith (+) s (tail s)) == (take 10 $ (tail . tail) s)
```

*Soluție:*

Fibonacci

4. Scrieți în Haskell o funcție care realizează produsul cartezian a două mulțimi oarecare (liste) A și B. Utilizați facilitățile oferite de limbaj. Care este tipul funcției create?

*Soluție:*

```
cart a b = [(x, y) | x <- a, y <- b]
cart :: [t] -> [t1] -> [(t, t1)]
```

5. Construiți în Haskell fluxul puterilor lui 2. 6. Construiți în Haskell fluxul puterilor lui 4.

*Soluție:*

```
fluxus = 1 : map (* 2) fluxus
```

*Soluție:*

```
fluxus = 1 : map (* 4) fluxus
```

7. Scrieți în Haskell o funcție care realizează produsul cartezian a două mulțimi oarecare (liste) A și B. Utilizați facilitățile oferite de limbaj. Care este tipul funcției create?

*Soluție:*

```
cart a b = [(x, y) | x <- a, y <- b]
cart :: [t] -> [t1] -> [(t, t1)]
```

5. Câte dintre cele trei adunări se vor realiza în evaluarea expresiei Haskell de mai jos?

Justificați!

```
let selector x y z = x in selector (1 + 2) (2 + 3) (3 + 4)
```

*Soluție:*

Una – (1 + 2), pentru că evaluarea este lenșă și parametrii y și z nu sunt evaluați.

6. Folosiți list comprehensions pentru a produce fluxul listelor formate din primii 5 multipli ai fiecărui număr natural:

```
[[1,2,3,4,5], [2,4,6,8,10], [3,6,9,12,15], [4,8,12,16,20] ...] .
```

*Soluție:*

```
[take 5 [m | m <- [n..], mod m n == 0] | n <- [1..]]
```

6. Folosiți list comprehensions pentru a produce fluxul listelor de divizori pentru numerele naturale: [[1], [1, 2], [1, 3], [1, 2, 4], [1, 5], [1, 2, 3, 6] ...] .

*Soluție:*

```
[[d | d <- [1..n], mod n d == 0] | n <- [1..]]
```

Scrieți o funcție sumPairs care primește două liste de numere și returnează o listă care conține suma elementelor de pe aceeași poziție din cele două liste.

```
sumPairs :: [Int] -> [Int] -> [Int]
sumPairs [] _ = []
sumPairs _ [] = []
sumPairs (x:xs) (y:ys) = (x + y) : sumPairs xs ys
```

Implementați o funcție applyTwice care primește o funcție f și un argument x și returnează rezultatul aplicării funcției f de două ori pe x.

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

Definiți o funcție isPalindrome care verifică dacă un sir de caractere este palindrom.

```
isPalindrome :: String -> Bool
isPalindrome s = s == reverse s
```

Implementați o funcție mergeSort care sortează o listă de numere folosind algoritmul Merge Sort.

```
mergeSort :: Ord a => [a] -> [a]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs =
    merge (mergeSort firstHalf) (mergeSort secondHalf)
    where
        merge [] ys = ys
        merge xs [] = xs
        merge (x:xs) (y:ys)
            | x <= y = x : merge xs (y:ys)
            | otherwise = y : merge (x:xs) ys
        (firstHalf, secondHalf) = splitAt (length xs `div` 2) xs
```

Scrieți o funcție factorial care calculează factorialul unui număr dat.

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Definiți o funcție flatten care primește o listă de liste și returnează o singură listă cu toate elementele inițiale.

```
flatten :: [[a]] -> [a]
```

```
flatten [] = []
```

```
flatten (x:xs) = x ++ flatten xs
```

Implementați o funcție fibonacci care calculează al n-lea număr Fibonacci.

```
fibonacci :: Int -> Int
```

```
fibonacci 0 = 0
```

```
fibonacci 1 = 1
```

```
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

Scrieți o funcție isPrime care verifică dacă un număr dat este prim.

```
isPrime :: Int -> Bool
```

```
isPrime n
```

```
| n <= 1 = False
```

```
| otherwise = all (\x -> n `mod` x /= 0) [2..isqrt n]
```

```
where
```

```
isqrt = floor . sqrt . fromIntegral
```

Definiți o funcție lastElement care returnează ultimul element dintr-o listă.

```
lastElement :: [a] -> a
```

```
lastElement [x] = x
```

```
lastElement (_:xs) = lastElement xs
```

Implementați o funcție map' care primește o funcție și o listă și aplică funcția pe fiecare element din listă.

```
map' :: (a -> b) -> [a] -> [b]
```

```
map' _ [] = []
```

```
map' f (x:xs) = f x : map' f xs
```

Scrieți o funcție length' care calculează lungimea unei liste.

```
length' :: [a] -> Int
```

```
length' [] = 0
```

```
length' (_:xs) = 1 + length' xs
```

Definiți o funcție maximum' care returnează elementul maxim dintr-o listă de numere.

```
maximum' :: Ord a => [a] -> a
```

```
maximum' [x] = x
```

```
maximum' (x:xs) = max x (maximum' xs)
```

Implementați o funcție takeWhile' care primește o funcție predicat și o listă și returnează prefixul listei format din elementele care satisfac predicatul.

```
takeWhile' :: (a -> Bool) -> [a] -> [a]
```

```
takeWhile' [] = []
```

```
takeWhile' p (x:xs)
```

```
| p x = x : takeWhile' p xs
```

```
| otherwise = []
```

Scriți o funcție everyNth care primește un număr n și o listă și returnează o nouă listă conținând doar elementele de pe pozițiile multiplu de n.

```
everyNth :: Int -> [a] -> [a]
```

```
everyNth n xs = [x | (i, x) <- zip [1..] xs, i `mod` n == 0]
```

Definiți o funcție isSorted care verifică dacă o listă de numere este sortată în ordine crescătoare.

```
isSorted :: Ord a => [a] -> Bool
```

```
isSorted [] = True
```

```
isSorted [_] = True
```

```
isSorted (x:y:xs) = x <= y && isSorted (y:xs)
```

Implementați o funcție rotate care primește un număr n și o listă și returnează lista rotită la stânga de n ori.

```
rotate :: Int -> [a] -> [a]
```

```
rotate [] = []
```

```
rotate n xs = drop n xs ++ take n xs
```

Scriți o funcție groupBy' care primește o funcție de echivalență și o listă și grupează elementele consecutive echivalente conform funcției de echivalență.

```
groupBy' :: (a -> a -> Bool) -> [a] -> [[a]]
```

```
groupBy' [] = []
```

```
groupBy' f (x:xs) = (x : takeWhile (f x) xs) : groupBy' f (dropWhile (f x) xs)
```

Definiți o funcție `interleave` care primește două liste și le intercalează elementele alternativ.

```
interleave :: [a] -> [a] -> [a]
```

```
interleave [] ys = ys
```

```
interleave xs [] = xs
```

```
interleave (x:xs) (y:ys) = x : y : interleave xs ys
```

Implementați o funcție `partition` care primește o funcție predicat și o listă și returnează o pereche de liste, una conținând elementele care satisfac predicatul și cealaltă conținând elementele care nu îl satisfac.

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
```

```
partition _ [] = ([], [])
```

```
partition p (x:xs)
```

```
| p x = (x : ys, zs)
```

```
| otherwise = (ys, x : zs)
```

```
where
```

```
(ys, zs) = partition p xs
```

Scrieți o funcție `transpose` care primește o listă de liste și returnează matricea transpusă.

```
transpose :: [[a]] -> [[a]]
```

```
transpose [] = []
```

```
transpose ([]:_)= []
```

```
transpose rows = map head rows : transpose (map tail rows)
```

Definiți o funcție `isSublist` care verifică dacă o listă este sublista unei alte liste.

```
isSublist :: Eq a => [a] -> [a] -> Bool
```

```
isSublist _ [] = False
```

```
isSublist xs ys
```

```
| take (length xs) ys == xs = True
```

```
| otherwise = isSublist xs (tail ys)
```

Implementați o funcție `insertionSort` care sortează o listă de numere folosind algoritmul Insertion Sort.

```
insertionSort :: Ord a => [a] -> [a]
```

```
insertionSort xs = foldr insert [] xs
```

where

```
insert x [] = [x]
```

```
insert x (y:ys)
```

```
  | x <= y = x : y : ys
```

```
  | otherwise = y : insert x ys
```

Scrieți o funcție isEven care verifică dacă un număr dat este par.

```
isEven :: Int -> Bool
```

```
isEven n = n `mod` 2 == 0
```

Definiți o funcție nub care elimină dupicații consecutive dintr-o listă.

```
nub :: Eq a => [a] -> [a]
```

```
nub [] = []
```

```
nub [x] = [x]
```

```
nub (x:y:xs)
```

```
  | x == y = nub (y:xs)
```

```
  | otherwise = x : nub (y:xs)
```

Implementați o funcție splitAt' care primește un index n și o listă și returnează o pereche de liste, una conținând primele n elemente și cealaltă conținând restul.

```
splitAt' :: Int -> [a] -> ([a], [a])
```

```
splitAt' n xs = (take n xs, drop n xs)
```



9. Construiți în Prolog un predicat `up(+L, -LUp)` care produce în `LUp` o listă cu următoarele proprietăți:

- primul element din `LUp` este același cu primul element din lista `L`.
- următorul element din `LUp` este următorul element din `L`, mai mare decât primul.
- următorul element din `LUp` este următorul element din `L`, mai mare decât anteriorul din `LUp`, etc.

Exemplu: `up([5, 6, 3, 4, 8, 5, 9, 2], LUp)` leagă `LUp` la `[5, 6, 8, 9]`.

*Soluție:*

```
% up(+L, -LUp)
up([H|T], [H|TUp]) :- upAux(H, T, TUp).
upAux(_, [], []).
% H face parte din secvența crescătoare:
upAux(CurrentMax, [H|T], [H|TUp]) :- H > CurrentMax, upAux(H, T, TUp).
% H nu face parte din secvența crescătoare:
upAux(CurrentMax, [H|T], TUp) :- H =\= CurrentMax, upAux(CurrentMax, T, TUp).
```

Alternativ, pot evita comparația din ultima regulă, dacă folosesc ! după comparația din penultima regulă.

9. Construiți în Prolog un predicat `dn(+L, -LDown)` care produce în `LDown` o listă cu următoarele proprietăți:

- primul element din `LDown` este același cu primul element din lista `L`.
- următorul element din `LDown` este următorul element din `L`, mai mic decât primul.
- următorul element din `LDown` este următorul element din `L`, mai mic decât anteriorul din `LDown`, etc.

Exemplu: `dn([5, 6, 3, 4, 8, 5, 9, 2], LDown)` leagă `LDown` la `[5, 3, 2]`.

*Soluție:*

```
% dn(+L, -LDown)
dn([H|T], [H|TDown]) :- dnAux(H, T, TDown).
dnAux(_, [], []).
% H face parte din secvența crescătoare:
dnAux(CurrentMin, [H|T], [H|TDown]) :- H < CurrentMin, dnAux(H, T, TDown).
% H nu face parte din secvența crescătoare:
dnAux(CurrentMin, [H|T], TDown) :- H =\= CurrentMin, dnAux(CurrentMin, T, TDown).
```

Alternativ, pot evita comparația din ultima regulă, dacă folosesc ! după comparația din penultima regulă.

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.
```

8. Implementați predicatul `gen(+Sample, +Length, -Output)`, care produce în `Output` o listă de lungime `Length` care constă din repetări ale listei `Sample`.

Exemplu: `g([1, 2, 3], 10, X)` leagă `X` la `[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]`.

Explicați cum funcționează implementarea.

*Soluție:*

```
g(_, 0, []).  
g([E|T], Len, [E|R]) :- Len > 0, Len1 is Len - 1, append(T, [E], T1), g(T1, Len1, R).
```

9. Folosiți o singură dată unul dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `minmax(+L, -Min, -Max)` care leagă `Min`, respectiv `Max`, la elementul minim, respectiv maxim, al listei. Nu utilizați recursivitate explicită. Explicați cum funcționează soluția.

*Soluție:*

```
minmax(L, Min, Max) :- member(Min, L), member(Max, L),  
forall(member(X, L), (X >= Min, X =< Max)).
```

8. Implementați predicatul `a(+L1, +L2, +L3, -L)` care primește 3 liste și produce în `L` o listă de tripluri cu elementele ce corespund din cele 3 liste. Lista `L` are lungimea celei mai scurte liste dintre `L1`, `L2` și `L3`. Exemplu: `a([1, 2], [a, b, c], [x, y, z, t], X)` leagă `X` la `[(1, a, x), (2, b, y)]`.

Explicați cum funcționează implementarea.

*Soluție:*

```
a([], _, _, []).  
a(_, [], _, []).  
a(_, _, [], []).  
a([H1|L1], [H2|L2], [H3|L3], [(H1, H2, H3) | L123]) :- a(L1, L2, L3, L123).
```

9. Folosiți unul sau mai multe dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `secondMin(+L, -X)` care găsește al doilea cel mai mic element din lista `L` (care conține cel puțin 2 elemente și nu conține duplicate). Nu utilizați recursivitate explicită. Explicați cum funcționează soluția.

*Soluție:*

```
secondMin(L, X) :- member(X, L), findall(Y, (member(Y, L), Y < X), [_]).
```

8. Implementați predicatul `mul(+Lin, -Lout)` care primește o listă de numere ca prim argument și leagă al doilea argument la lista acelor numere din prima listă care sunt divizibile cu toate celelalte numere care le urmează.

Exemplu: `mul([24, 4, 12, 6, 3, 2], X)` leagă `X` la `[24, 12, 6, 2]`.

Explicați cum funcționează implementarea.

*Soluție:*

```
check([], []).  
check(X, [H|T]) :- mod(X, H) == 0, check(X, T).  
mul([], []).  
mul([H|LIn], [H|LOut]) :- check(H, LIn), !, mul(LIn, LOut).  
mul([_|LIn], LOut) :- mul(LIn, LOut).
```

9. Folosiți unul sau mai multe dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `secondMax(+L, -X)` care găsește al doilea cel mai mare element din lista L (care conține cel puțin 2 elemente și nu conține duplicate). Nu utilizați recursivitate explicită. Explicați cum funcționează soluția.

*Soluție:*

```
secondMin(L, X) :- member(X, L), findall(Y, (member(Y, L), Y < X), [_]).
```

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]`

8. Se dă programul Prolog:

```
p(_, [], []).
p(A, [A|B], B) :- !.
p(A, [B|C], [B|D]) :- p(A, C, D).
```

Ce relație există între cele 3 valori X, Y, Z, dacă `p(X, Y, Z)` este adevărat?

*Soluție:*

Este predicatul `select`, iar dacă primul argument este nelegat face `select` la primul element. Predicatul `select(X, Y, Z)` este adevărat dacă X este un element din lista Y, iar Z este exact lista Y, în afară de elementul X.

8. Scrieți un predicat Prolog `diff(A, B, R)` care leagă R la diferența mulțimilor (reprzentate ca liste) A și B.

*Soluție:*

```
intersect(A, B, R) :- findall(X, (member(X, A), member(X, B)), R).
```

8. Scrieți un predicat Prolog `intersect(A, B, R)` care leagă R la intersecția mulțimilor (reprzentate ca liste) A și B.

*Soluție:*

```
diff(A, B, R) :- findall(X, (member(X, A), \+ member(X, B)), R).
```

7. Implementați în Prolog predicatul `x(L, A, B, N)` care determină, pentru o listă L, numărul N de elemente care sunt mai mari decât A și mai mici decât B. Nu folosiți recursivitate explicită.

*Soluție:*

```
x(L, A, B, N) :- findall(X, (member(X, L), X > A, X < B), S), length(S, N).
```

7. Implementați în Prolog predicatul `x(L, A, B, N)` care determină, pentru o listă L, numărul N de elemente care nu sunt mai mari decât A și mai mici decât B. Nu folosiți recursivitate explicită.

*Soluție:*

```
x(L, A, B, N) :- findall(X, (member(X, L), (X < A; X > B)), S), length(S, N).
```

7. Implementați în Prolog predicatul `x(L, M)` care determină, pentru o listă L, M, maximul listei. Nu folosiți recursivitate explicită.

*Soluție:*

```
x(L, M) :- member(M, L), forall(member(E, L), X > E).
```

7. Implementați în Prolog predicatul `x(L, M)` care determină, pentru o listă L, M, minimul listei. Nu folosiți recursivitate explicită.

*Soluție:*

```
x(L, M) :- member(M, L), forall(member(E, L), X < E).
```

10. Care este efectul aplicării predicatului `p` asupra listelor `L1` și `L2` (la ce este legat argumentul `R` în apelul `p(L1, L2, R)` ?):

```
p(A, [], A). p(A, [E|T], [E|R]) :- p(A, T, R).
```

*Soluție:*

```
R = L2 ++ L1
```

10. Care este efectul aplicării predicatului `p` asupra listelor `L1` și `L2` (la ce este legat argumentul `R` în apelul `p(L1, L2, R)` ?):

```
p([], A, A). p([E|T], A, [E|R]) :- p(T, A, R).
```

*Soluție:*

```
R = L1 ++ L2
```

10. Implementați în Prolog un predicat având semnătura `mapF(+L, -LO)`, care să fie echivalent cu expresia Haskell (`map f`), știind că există deja definit un predicat `f(+X, -XO)`.

*Soluție:*

```
mapF(L, LO):- findall(XO, (member(X, L), f(X, XO)), LO).
```

11. Câte soluții are interogarea `p(L, [1, 2, 3])` în condițiile în care avem definiția de mai jos? Ce formă au aceste soluții?

```
p(D, [A, B, C]) :- member(A, D), member(B, D), member(C, D).
```

*Soluție:*

O infinitate. Soluțiile sunt liste care conțin 1, 2 și 3 (în ordine) și un număr  $\geq 0$  de elemente neinstantiate.

10. Implementați în Prolog un predicat având semnătura `filterF(+L, -LO)`, care să fie echivalent cu expresia Haskell (`filter f`), știind că există deja definit un predicat `f(+X)`.

*Soluție:*

```
filterF(L, LO):- findall(X, (member(X, L), f(X)), LO).
```

11. Câte soluții are interogarea `p([1, 2, 3], L)` în condițiile în care avem definiția de mai jos? Ce formă au aceste soluții?

```
p(D, [A, B, C]) :- member(A, D), member(B, D), member(C, D).
```

*Soluție:*

27. Sunt toate combinațiile de 1,2,3, inclusiv în care un element apare de mai multe ori.

10. Ce rezultat are în Prolog evaluarea lui `p(L, X)`, cu `L` o listă și `X` nelegat:

```
r([], _).
```

```
r([H|T], X) :- member(H, X), r(T, X).
```

```
p(L, X) :- length(L, N), length(X, N), r(L, X).
```

*Soluție:*

Lista `X` are aceeași lungime ca și `L` și aceeași membri, în orice ordine (diversele soluții pentru `X` sunt permutările listei `L`).

11. Scrieți un predicat Prolog `up` (și eventual predicate ajutătoare) care identifică secvențele (cel puțin două elemente) strict crescătoare dintr-o listă. Exemplu:

```
up([5, 1, 2, 3, 2, 3, 1, 1, 0, 9, 10], LS) → 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).
```

10. Scrieți un predicat Prolog `down` (și eventual predicate ajutătoare) care identifică secvențele (cel puțin două elemente) strict descrescătoare dintr-o listă. Exemplu:  
`down([5, 1, 2, 3, 2, 1, 1, 10, 9, 10], LS) → LS = [5, 1, 3, 2, 1, 10, 9]`

*Soluție:*

```
down([], []).  
down([H, H1 | T], [H, H1 | LS]) :- H1 < H, !, down(T, H1, LS).  
down([_ | T], LS) :- down(T, LS).  
down([], _, []) :- !.  
down([H | T], E, [H | LS]) :- H < E, !, down(T, H, LS).  
down(L, _, LS) :- down(L, LS).
```

11. Ce rezultat are în Prolog evaluarea lui `transformer(L, X)`, cu L o listă și X nelegat:

```
processor([], _).  
processor([H|T], X) :- member(H, X), processor(T, X).  
transformer(L, X) :- length(L, N), length(X, N), processor(L, X).
```

*Soluție:*

Lista X are aceeași lungime ca și L și aceeași membri, în orice ordine (diversele soluții pentru X sunt permutările listei L).

9. Scrieți predicatul `filter(+L, +T, -LF)` în Prolog care ia o listă și o filtrează, întorcând în LF doar acele elemente mai mari (strict) decât pragul T, fără a utiliza recursivitate explicită. Exemplu: `filter([1,5,9,3,2,6,3,8], 5, LF)` leagă LF la [9, 6, 8].

*Soluție:*

```
filter(L, T, LF) :- findall(X, (member(X, L), X > T), LF).
```

10. Realizați un predicat Prolog care elimină duplicatele dintr-o listă.

*Soluție:*

```
nodups([], []).  
nodups([H|T], [H|LO]) :- findall(X, (member(X, T), X \= H), T1),  
nodups(T1, LO).  
sau  
rem(_, [], []).  
rem(X, [X|L], LO) :- rem(X, L, LO), !.  
rem(X, [H|L], [H|LO]) :- rem(X, L, LO).  
nodups([], []).  
nodups([H|T], [H|LO]) :- rem(H, T, T1), nodups(T1, LO).
```



## Semester 1 Haskell

add :: !Int → Int → Int

inc = add 1

map :: (a → b) → [a] → [b]

map f [] = []

map f (x:xs) = f x : map f xs

(++) :: [a] → [a] → [a]

[] ++ ys = ys

(x:xs) ++ ys = x:(xs ++ ys)

(.) :: (b → c) → (a → b) → (a → c)

f . g = \x → f(g x)

ones = 1: ones

length n = n : length (n+1)

squares = map (^2) (numbers 0)

fib = \b : [a] → b | (a,b) ← zip fib (map fib fib)

zip (x:xs) (y:ys) = (x,y) : zip xs ys

zip xs ys = []

head (x:xs) = x

head [] = error ; head :: [a] → a

length :: [a] → Int

length [] = 0

length (x:xs) = 1 + length xs

tail :: [a] → [a]

tail (x:xs) = xs.

$x \text{ 'elem' } c = \text{False}$

$x \text{ 'elem' } (y: \alpha) = x == y \wedge (\exists \alpha \text{ 'elem' } y)$

$\text{elem} : (\text{Eq } \alpha) \Rightarrow \alpha \rightarrow [\alpha] \rightarrow \text{Bool}$

$(+), (-), (\times) : (\text{Num } \alpha) \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$

$\text{negate}, \text{abs} : (\text{Num } \alpha) \Rightarrow \alpha \rightarrow \alpha$ .

nearest, round, floor, ceiling, numerator, denominator, conjugate, hollow, sqrt,  
tuke, drop, filter

1. Funcția `id`:	13. Funcția `head`:	product :: Num a => [a] -> a
id :: a -> a	head :: [a] -> a	27. Funcția `elemIndex`:
2. Funcția `const`:	14. Funcția `tail`:	elemIndex :: Eq a => a -> [a] -> Maybe Int
const :: a -> b -> a	tail :: [a] -> [a]	28. Funcția `elemIndices`:
3. Funcția `flip`:	15. Funcția `init`:	elemIndices :: Eq a => a -> [a] -> [Int]
flip :: (a -> b -> c) -> b -> a -> c	init :: [a] -> [a]	29. Funcția `find`:
4. Funcția `(.)` (compoziție de funcții):	16. Funcția `last`:	find :: (a -> Bool) -> [a] -> Maybe a
(.) :: (b -> c) -> (a -> b) -> a -> c	last :: [a] -> a	30. Funcția `partition`:
5. Funcția `(\$)`: (aplicare de funcții)	17. Funcția `reverse`:	partition :: (a -> Bool) -> [a] -> ([a], [a])
(\$) :: (a -> b) -> a -> b	reverse :: [a] -> [a]	EXERCITII
6. Funcția `map`:	18. Funcția `length`:	Exemplu 1:
map :: (a -> b) -> [a] -> [b]	length :: [a] -> Int	-- Exemplu 1
7. Funcția `filter`:	19. Funcția `take`:	f1 x y = (x > y, x + y)
filter :: (a -> Bool) -> [a] -> [a]	take :: Int -> [a] -> [a]	Sintetizare:
8. Funcția `foldr`:	20. Funcția `drop`:	f1 :: Ord a => a -> a -> (Bool, a)
foldr :: (a -> b -> b) -> b -> [a] -> b	drop :: Int -> [a] -> [a]	x :: Ord a => a
9. Funcția `foldl`:	21. Funcția `elem`:	y :: Ord a => a
foldl :: (b -> a -> b) -> b -> [a] -> b	elem :: Eq a => a -> [a] -> Bool	(>) :: Ord a => a -> a -> Bool
10. Funcția `zip`:	22. Funcția `notElem`:	(+) :: Num a => a -> a -> a
zip :: [a] -> [b] -> [(a, b)]	notElem :: Eq a => a -> [a] -> Bool	Exemplu 2:
11. Funcția `zipWith`:	23. Funcția `maximum`:	-- Exemplu 2
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]	maximum :: Ord a => [a] -> a	f2 x y = x + (head y)
12. Funcția `concat`:	24. Funcția `minimum`:	Sintetizare:
concat :: [[a]] -> [a]	minimum :: Ord a => [a] -> a	f2 :: Num a => a -> [a] -> a
	25. Funcția `sum`:	x :: Num a => a
	sum :: Num a => [a] -> a	
	26. Funcția `product`:	

$y :: [a]$	$f6 x y = x ++ y$	$foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a]$
$head :: [a] \rightarrow a$	Sintetizare:	$\rightarrow b$
$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$	$f6 :: [a] \rightarrow [a] \rightarrow [a]$	Exemplu 10:
Exemplu 3:	$x :: [a]$	$f10 f g x = f(g x)$
-- Exemplu 3	$y :: [a]$	Sintetizare:
$f3 f g x = f(g x)(g(x + 1))$	$(++) :: [a] \rightarrow [a] \rightarrow [a]$	$f10 :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$
Sintetizare:	Exemplu 7:	$f :: a \rightarrow b$
$f3 :: (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow a) \rightarrow d \rightarrow c$	$f7 xs = \text{filter } (\lambda x \rightarrow x > 0) xs$	$g :: c \rightarrow a$
$f :: a \rightarrow b \rightarrow c$	Sintetizare:	$x :: c$
$g :: d \rightarrow a$	$f7 :: (\text{Num } a, \text{Ord } a) \Rightarrow [a] \rightarrow [a]$	Exemplu 11:
$x :: d$	$xs :: (\text{Num } a, \text{Ord } a) \Rightarrow [a]$	$f11 xs = \text{reverse } (\text{tail } xs)$
$(+) :: \text{Num } d \Rightarrow d \rightarrow d \rightarrow d$	$\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$	Sintetizare:
Exemplu 4:	$(>) :: \text{Ord } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$	$f11 :: [a] \rightarrow [a]$
-- Exemplu 4	Exemplu 8:	$xs :: [a]$
$f4 xs = \text{length } xs > 0$	$f8 xs = \text{map } (\lambda x \rightarrow x * x) xs$	$\text{reverse} :: [a] \rightarrow [a]$
Sintetizare:	Sintetizare:	$\text{tail} :: [a] \rightarrow [a]$
$f4 :: [a] \rightarrow \text{Bool}$	$f8 :: \text{Num } b \Rightarrow [b] \rightarrow [b]$	Exemplu 12:
$xs :: [a]$	$xs :: \text{Num } b \Rightarrow [b]$	$f12 f g x y = f(g(x, y))$
$\text{length} :: [a] \rightarrow \text{Int}$	$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$	Sintetizare:
$(>) :: \text{Ord } a \Rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$	$(*) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$	$f12 :: ((a, b) \rightarrow c) \rightarrow (d \rightarrow (a, b)) \rightarrow d \rightarrow c$
Exemplu 5:	Exemplu 9:	$f :: (a, b) \rightarrow c$
$f5 x = [x, x + 1, x + 2]$	$f9 f g xs = \text{foldr } (\lambda x acc \rightarrow f(g x) acc) [] xs$	$g :: d \rightarrow (a, b)$
Sintetizare:	Sintetizare:	$x :: d$
$f5 :: \text{Num } a \Rightarrow a \rightarrow [a]$	$f9 :: (b \rightarrow a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow [c] \rightarrow b$	$y :: d$
$x :: \text{Num } a \Rightarrow a$	$f :: b \rightarrow a \rightarrow b$	Exemplu 13:
$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$	$g :: c \rightarrow a$	$f13 x = x == \text{reverse } x$
Exemplu 6:	$xs :: [c]$	Sintetizare:

f13 :: Eq a => [a] -> Bool

x :: Eq a => [a]

(==) :: Eq a => a -> a -> Bool

reverse :: [a] -> [a]

Exemplu 14:

f14 f g x = f (g x) x

Sintetizare:

f14 :: (a -> b -> c) -> (d -> a) -> d -> c

f :: a -> b -> c

g :: d -> a

x :: d

Exemplu 15:

f15 x y = zipWith (\a b -> (a, b)) x y

Sintetizare:

f15 :: [a] -> [b] -> [(a, b)]

x :: [a]

y :: [b]

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

Exemplu 16:

f16 xs = sum [1 | \_ <- xs]

Sintetizare:

f16 :: Num a => [b] -> a

xs :: [b]

sum :: Num a => [a] -> a

Exemplu 17:

f17 xs = foldl (\acc x -> if even x then x:acc else acc) [] xs

Sintetizare:

f17 :: Integral a => [a] -> [a]

xs :: Integral a => [a]

foldl :: (b -> a ->

b) -> b -> [a] -> b

even :: Integral a => a -> Bool

Exemplu 18:

f18 x y = zip (x ++ [y]) [y]

Sintetizare:

f18 :: [a] -> a -> [(a, a)]

x :: [a]

y :: a

zip :: [a] -> [b] -> [(a, b)]

(++) :: [a] -> [a] -> [a]

Exemplu 19:

f19 x y = replicate x y

Sintetizare:

f19 :: Int -> a -> [a]

x :: Int

y :: a

replicate :: Int -> a -> [a]

Exemplu 20:

f20 x = map (\y -> y \* y) x

Sintetizare:

f20 :: Num b => [b] -> [b]

x :: Num b => [b]

map :: (a -> b) -> [a] -> [b]

(\*) :: Num a => a -> a -> a

Exemplu 21:

f21 x = map (\y -> (y, y \* y)) x

Sintetizare:

f21 :: Num b => [a] -> [(a, b)]

x :: [a]

map :: (a -> b) -> [a] -> [b]

(\*) :: Num a => a -> a -> a

Exemplu 22:

f22 f x y = f (x + y)

Sintetizare:

f22 :: (Num a, Num b) => (a -> b) -> a -> a -> b

f :: Num a => a -> b

x :: Num a => a

y :: Num a => a

(+) :: Num a => a -> a -> a

Exemplu 23:

f23 f x y = f (f x) (f y)

Sintetizare:

f23 :: (a -> b) -> a -> a -> b

f :: a -> b

x :: a

y :: a

Exemplu 24:

f24 x = zip x (tail x)

Sintetizare:	Exemplu 3:	(+) :: Num a => a -> a -> a
f24 :: [a] -> [(a, a)]	f3 x y = x : y : []	(-) :: Num a => a -> a -> a
x :: [a]	Sintetizare:	Exemplu 7:
zip :: [a] -> [b] -> [(a, b)]	f3 :: a -> a -> [a]	f7 x y = x : y : []
tail :: [a] -> [a]	x :: a	Sintetizare:
Exemplu 25:	y :: a	f7 :: a -> a -> [a]
f25 f g x = (f x, g x)	Exemplu 4:	x :: a
Sintetizare:	f4 f g x = f (g (g x))	y :: a
f25 :: (a -> b) -> (a -> c) -> a -> (b, c)	Sintetizare:	Exemplu 8:
f :: a -> b	f4 :: (a -> b) -> (c -> a) -> c -> b	f8 f g x y = f (g (x, y))
g :: a -> c	f :: a -> b	Sintetizare:
x :: a	g :: c -> a	f8 :: ((a, b) -> c) -> (d -> (a, b)) -> d -> c
Exemplu 1:	x :: c	f :: (a, b) -> c
f1 x = filter (\y -> y > 0) x	Exemplu 5:	g :: d -> (a, b)
Sintetizare:	f5 x = x ++ reverse x	x :: d
f1 :: (Num a, Ord a) => [a] -> [a]	Sintetizare:	y :: d
x :: (Num a, Ord a) => [a]	f5 :: [a] -> [a]	Exemplu 9:
filter :: (a -> Bool) -> [a] -> [a]	x :: [a]	f9 x y = replicate x y
(>) :: Ord a => a -> a -> Bool	(++) :: [a] -> [a] -> [a]	Sintetizare
Exemplu 2:	reverse :: [a] -> [a]	f9 :: Int -> a -> [a]
f2 f x y = (f x, f y)	Exemplu 6:	x :: Int
Sintetizare:	f6 f x y = f (x + y) (x - y)	y :: a
f2 :: (a -> b) -> a -> a -> (b, b)	Sintetizare:	replicate :: Int -> a -> [a]
f :: a -> b	f6 :: (Num a, Num b) => (a -> a -> c) -> a -> b -> c	Exemplu 10:
x :: a	f :: (Num a, Num b) => a -> a -> c	f10 f x y = (f x y, f y x)
y :: a	x :: (Num a, Num b) => a	
	y :: (Num a, Num b) => b	

Sintetizare:	$y :: b$	$x :: d$
$f10 :: (a \rightarrow b \rightarrow c) \rightarrow a \rightarrow b \rightarrow (c, c)$	Exemplu 14: $f14 f x = f(f x)$	$y :: d$
$f :: a \rightarrow b \rightarrow c$	Sintetizare:	Exemplu 18: $f18 f x y = f(f x)(f y)$
$x :: a$	$f14 :: (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow b$	Sintetizare:
$y :: b$	$f :: a \rightarrow a \rightarrow b$	$f18 :: (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a \rightarrow c$
Exemplu 11:	$x :: a$	$f :: a \rightarrow b \rightarrow c$
$f11 f g x y = f(g x)(g y)$	Exemplu 15: $f15 f x = f(f x)$	$x :: a \rightarrow a$
Sintetizare:	Sintetizare:	Exemplu 19: $f19 f x y = f(f(x, y))$
$f11 :: (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow a) \rightarrow d \rightarrow d \rightarrow c$	$f15 :: (a \rightarrow a) \rightarrow a \rightarrow a$	Sintetizare:
$f :: a \rightarrow b \rightarrow c$	$f :: a \rightarrow a$	$f19 :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$
$g :: d \rightarrow a$	$x :: a$	$f :: (a, b) \rightarrow c$
$x :: d$	Exemplu 16: $f16 f g x y = f(g x y)$	$x :: a$
$y :: d$	Sintetizare:	$y :: b$
Exemplu 12:	$f16 :: (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow e \rightarrow a) \rightarrow d \rightarrow e \rightarrow c$	Exemplu 20: $f20 f g x y = f(g x)(g y)$
$f12 f x y = f(f x)(f y)$	$f :: a \rightarrow b \rightarrow c$	Sintetizare:
Sintetizare:	$g :: d \rightarrow e \rightarrow a$	$f20 :: (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow a) \rightarrow (e \rightarrow a) \rightarrow d \rightarrow e \rightarrow c$
$f12 :: (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow a \rightarrow b$	$x :: d$	$f :: a \rightarrow b \rightarrow c$
$f :: a \rightarrow a \rightarrow b$	$y :: e$	$g :: d \rightarrow a$
$x :: a$	Exemplu 17: $f17 f g x y = f(g x)(g y)$	$x :: d$
$y :: a$	Sintetizare:	$y :: e$
Exemplu 13:	$f17 :: (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow a) \rightarrow d \rightarrow d \rightarrow c$	Exemplu 21: $f21 f g x y = f(g x y)(g y x)$
$f13 f x y = f(f x y) y$	$f :: a \rightarrow b \rightarrow c$	
Sintetizare:	$g :: d \rightarrow a$	
$f13 :: (a \rightarrow b \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$		
$f :: a \rightarrow b \rightarrow c$		
$x :: a$		

Sintetizare:	$y :: a$
$f21 :: (a \rightarrow a \rightarrow b) \rightarrow (c \rightarrow c \rightarrow b)$	Exemplu 25:
$a) \rightarrow c \rightarrow c \rightarrow b$	$f25 f g x y = f(g x)(g y)$
$f :: a \rightarrow a \rightarrow b$	Sintetizare:
$g :: c \rightarrow c \rightarrow a$	$f25 :: (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow a) \rightarrow (e \rightarrow b) \rightarrow d \rightarrow e \rightarrow c$
$x :: c$	$f :: a \rightarrow b \rightarrow c$
$y :: c$	$g :: d \rightarrow a$
Exemplu 22:	$x :: d$
$f22 f g x y = f(g x y)$	$y :: e$
Sintetizare:	
$f22 :: (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow e \rightarrow a) \rightarrow d \rightarrow e \rightarrow c$	
$f :: a \rightarrow b \rightarrow c$	
$g :: d \rightarrow e \rightarrow a$	
$x :: d$	
$y :: e$	
Exemplu 23:	
$f23 f x y = f(x, y)$	
Sintetizare:	
$f23 :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$	
$f :: (a, b) \rightarrow c$	
$x :: a$	
$y :: b$	
Exemplu 24:	
$f24 f x y = f(f x y)(f y x)$	
Sintetizare:	
$f24 :: (a \rightarrow a \rightarrow b) \rightarrow a \rightarrow a \rightarrow b$	
$f :: a \rightarrow a \rightarrow b$	
$x :: a$	