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 adevărată 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 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 `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 λ; 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 ((`a_i`v_i)) corp)` este echivalent cu `(lambda (`a_i`) 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?  (filter (compose ((curry <) e) abs) L))) L))
```
sau
```
(car (filter (λ(e) (null?  (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 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. 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 mulțimi (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 () ≡ [])
   `(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 () ≡ [])
   `(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)))` → `'((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)))` → `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
   ```