LAB 4

Sunday, March 20, 2022 6:43 PM

RECAPITULARE LAB 3

(1) apply => apply f '(l_1 l_2 ... l_m) =>

[f l_1 l_2 l_m]

=> apply extrage elem. dintr-o listă și le pasează funcției

(ex) ~~apply~~ map list '((1 2 3) (4 5 6)) =>

map list '(1 2 3) '(4 5 6) => '((1 4) (2 5) (3 6))

(ex) (~~apply~~ map + (~~apply~~ map list '((1 2 3) (4 5 6))))

=> map list '(1 2 3) '(4 5 6)

=> '((1 4) (2 5) (3 6))

=> map + '(1 4) '(2 5) '(3 6) =>

'(6 15)

=> mereu începem din interior

(ex) (~~apply~~ + (append '(1 2 3) '(1 2 3))) => + '(1 2 3 1 2 3) => 1 2

(2) foldl f v L_1 L_2 ... L_m

- f = λ x_1 x_2 ... x_m acc

  => f este o funcție cu param x_1 = elem. curent din L_1

  x_2 = elem. curent din L_2

  ---

  x_m = elem. curent din L_m

  acc = acumulatorul

  ↳ primă de la v

  ↳ acumulează rezultatul

- v = valoarea inițială
- L_1 L_2 ... L_m = liste

! Rezultatul final NU este mapărat o listă (ca la map și filter)

! Lista se parcurge ⟶ (de la stânga la dreapta)

! f îmi spune ce fac cu elem. curent din listă (x) și acc

Ex foldl (λ (x acc) (cons x acc)) '() '(1 2 3) =>

1 : [] -> 2 : [1] -> 3 : [2 1] => [3 2 1]

=> foldl cons '() '(1 2 3)

=> dacă am λ (x acc) -> f x acc, pot scrie direct f

- În general, putem plica de la o funcție λ (x acc), iar dacă vedem că param. sunt aceiași, putem renunța la λ și scriem direct f

Ex (foldl (λ (x1 x2 acc) (cons (+ x1 x2) acc)) '() '(1 2 3) '(4 5 6))

3 foldr f v L_1 L_2 ... L_n

-> similar cu foldl, doar că lista este parcursă ⟵ (de la dr. la stg)

foldl vs. foldr :

1. (foldl (λ (x acc) (cons (+ 1 x) acc)) '() '(1 2 3)) => '(4 3 2)
   - (foldr (λ (x acc) (cons (+ 1 x) acc)) '() '(1 2 3)) => '(2 3 4)
2. (foldl append '() '((1 2) (3 4))) => '(3 4 1 2)

L: [[1,2], [3,4]] -> [[3,4]] -> []

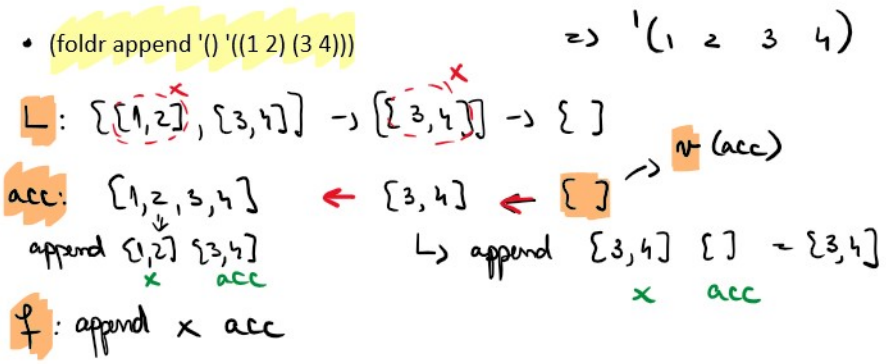
acc []: [1,2] -> [3,4,1,2] -> acc

└> append [1,2] [] => [1,2]

x acc

f = append x acc

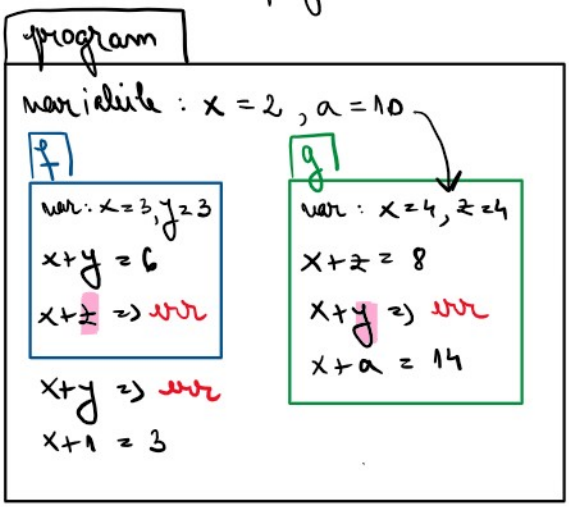
- (foldr append '() '((1 2) (3 4)))



LAB 4

DOMENIU DE VIZIBILITATE / SCOPE (al unei variabile)

=> unde în program variabila este vizibilă



=> fiecare variabilă este vizibilă în "cutia" ei

↓

Scope

program -> { x = 2, a = 10, f, g

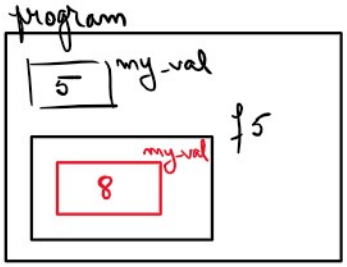
f -> { x = 3, y = 3, a = 10

g -> { x = 4, z = 4, a = 10

LEGARE / BINDING

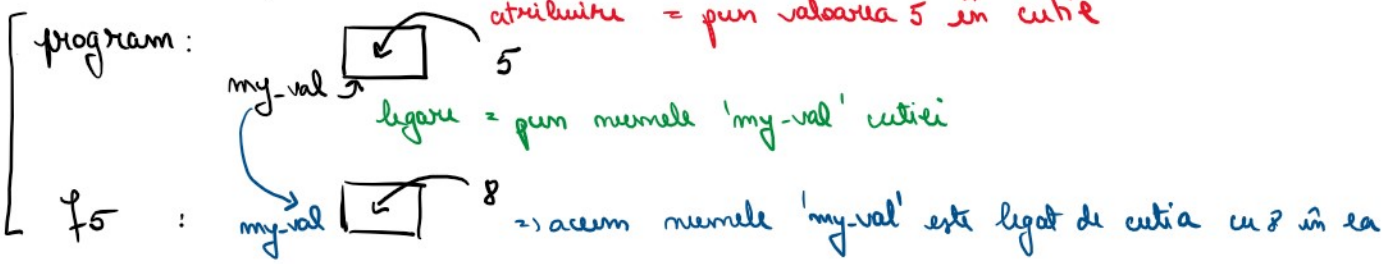
LEGARE vs. ATRIBUIRE

```
(define my-val 5)
;my-val

(define (f5 my-val) my-val)
;(f5 8)

(define (f6 x) my-val)
;(f6 1)
```



legare => legăm numele de cutie

atribuire => punem valoarea în cutie



- Pt. a fi mai ușor de înțeles, în Racket putem să ne gândim că 'my_val' este legat la un moment dat de conținutul cutiei (ex: valoarea 8)

LEGARE + SCOPE

În Racket, nu pot lega aceeași mereu la 2 entități diferite în același scope.

ex: (define x 5)
(define x 6)

Cum legăm variabile?

① **define** => legare globală

(define x 5) => x este legat la 5 și este vizibil global (în tot programul)

② **când apelăm o funcție** => (define (f x) (+ x 1)) => x se leagă la 2
(f 2)

③ **let** -> introduce un scope nou

SINTAXĂ:

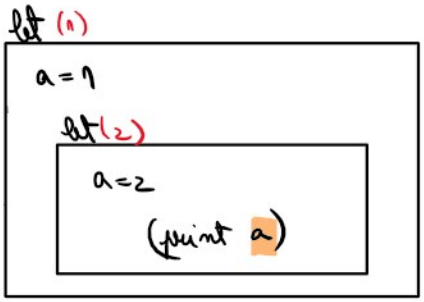
(let ((...) (...) ... (...)) ; definiții ex: (a 2) (b 5)
expr1 expr2 ... exprn) ; corpul let-ului ex: (+ x y) (list x y)
└> e întotdeauna let

-> **vizibilitate**: variabilele din let sunt vizibile DOAR în corpul let-ului

ex: (let ((x 2) (y 3)) (+ x y) (list x y)) => '(2 3)
└> rezultatul întors de let

=> x este legat la 2
y este legat la 3
} x și y sunt vizibile în (+ x y)

```
(let ([a 1]) ; prima legare
  (let ([a 2]) (print a)) )
```

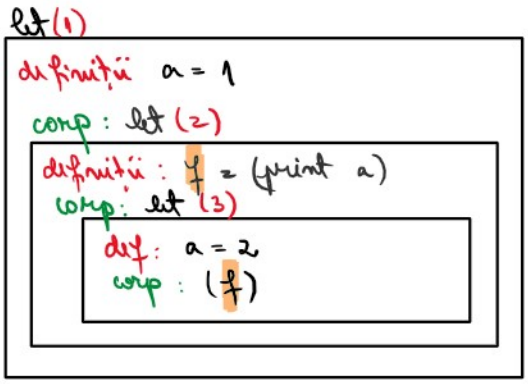


=> rezultatul întors de let 1 este let 2
=> rezultatul întors de let 2 este rezultatul lui (print a) = 2

LEGARE STATICĂ / DINAMICĂ

LEGARE STATICĂ → în Racket

```
(let ([a 1])
  (let ([f (lambda () (print a))])
    (let ([a 2]) (f)) ) )
```



=> se returnează (f)
f este legat cel mai aproape de (print a)
a este legat cel mai aproape de 1
=> afișează 1

LEGARE DINAMICĂ

→ ultima legare a lui a era la 2 => ar fi afișat 2

4) **let*** => la fel ca let, doar că var. sunt vizibile în { următoarele definiții în corp

5) **letrec**

ÎNCHIDERI FUNCȚIONALE (CLOSURE)

=> rezultatul produs de o λ-expresie

= { environment / context of the closure
  text of the λ-expr.

EXERCIȚII

② $(x_1\ y_1)$ $(x_2\ y_2)$

$\Rightarrow dist = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}$

$\Rightarrow$ formula finală $\begin{cases} x_1 = \\ x_2 = \\ y_1 = \\ y_2 = \end{cases}$

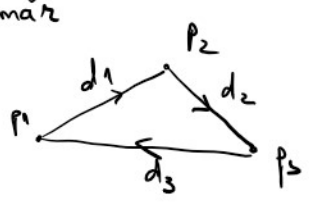
$\begin{cases} punct = (x_1\ y_1) = \text{o pereche de numere} \\ segment = (p_1\ p_2) = ((x_1\ y_1)\ (x_2\ y_2)) = \text{o pereche de puncte} \end{cases}$

① (get-line-segment) $\Rightarrow$ întoarce un segment

② (get-start-point segment) $\Rightarrow$ întoarce un punct (p_1)
(get-end-point segment) $\Rightarrow$ întoarce un punct (p_2)

③ $x_1 = \ldots$ $x_2 = \ldots$
$y_1 = \ldots$ $y_2 = \ldots$

④ (compute-perimetrer points) $\Rightarrow$ întoarce un număr; points $\Rightarrow$ listă de puncte $(p_1\ p_2 \ldots)$



perim $(p_1\ p_2\ p_3)$ = dist $(p_1\ p_2)$ +
dist $(p_2\ p_3)$ +
dist $(p_3\ p_1)$

first-point = p_1

L: $[p_1, p_2, p_3]$ -> $[p_2, p_3]$ -> $[p_3]$

acc 0: + d (p_1, p_2) -> + d (p_2, p_3) -> + d (p_3, fp)

$\begin{cases} perim\_helper\ [h]\ acc = acc + dist\ (h, fp) \\ perim\_helper\ (h_1 : h_2 : t) = perim\_helper\ t\ (acc + dist\ (h_1, h_2)) \end{cases}$

```
(let ([fp (car L)])
   (let perim-helper (....) ...))
```

HINT: pentru a verifica că o listă are un elem: (null? (cdr L))

(5) player -> poate mânca 1 sau 2

opponent -> poate mânca 2 sau 3

- player începe jocul
- Poate player să câștige => #t
  => #f

Mai poate să joace / să câștige dacă:

↳ player: (1) a mai rămas cel puțin o bomboană

(2) tura următoare nu câștigă opponent <=>
-> dacă mănâncă 1 bomboană => opponent nu câștigă (3)
-> dacă mănâncă 2 bomboane => oponent nu câștigă (4)

= 1 and 2 <=> 1 and (3 or 4)

↳ opponent: (1) au mai rămas cel puțin 2 bomboane

(2) tura următoare nu câștigă player <=>
->dacă mănâncă 2 bomboane => player nu câștigă (3)
->dacă mănâncă 3 bomboane -> player nu câștigă (4)

= 1 and (3 or 4)

=> Avem nevoie de 2 funcții { (player-can-play n) (opponent-can-play n) } -> întorc #t sau #f
(n = nr. de bomboane)

```
player-can-play n = (n >= 1) && ( ! opponent-can-play (n-1) ||
                                  ! opponent-can-play (n-2))

opponent-can-play n = (n >= 2) && (! player-can-play (n-2) ||
                                   ! player-can-play (n-3))
```

(6) split-at L pred => { takes elements from L as long as pred is #t => L_1
L_2 the rest of the list

3-secvente-max

ex: L= 1 2 3 0 4 2 1 0 1 1 1
S S
left mid right

• Ce trebuie să calculăm? => (max (sum left) (sum mid) (sum right))

(left rest1) = (split-at L pred) => păstrăm elem cât timp pred = #t
păstrăm elem cât timp != separator
=> pred = (λx -> x != separator)

(mid right) = (split-at rest1 pred)

(7) trebuie să legăm '+' la o nouă definiție locală => o reimplementare a lui '+' care concatenează 2 nr.

x y
22 33
↓ ↓ number -> string
'22' '33'
⇓ string-append
'2233'
↓ string -> number
2233