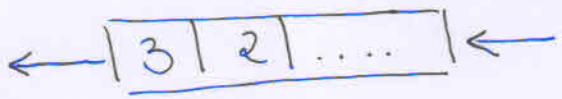
§5 Funkcje destruktywne

do tej pory: czyste funkcje matematyczne

```
> (define (f x) ... )
f
> (f 3)
7
> (f 3)
7
```

to czasami "niewygodne": queue



↳ trzeba modelować całą historię

Przykład: konto

"balance = 100"

```
> (withdraw 50)
50
> (withdraw 30)
20
```

do tego polecenie set!

(set! x w)

zmieni (przepisze) wartość zmiennej x do w w środowisku.

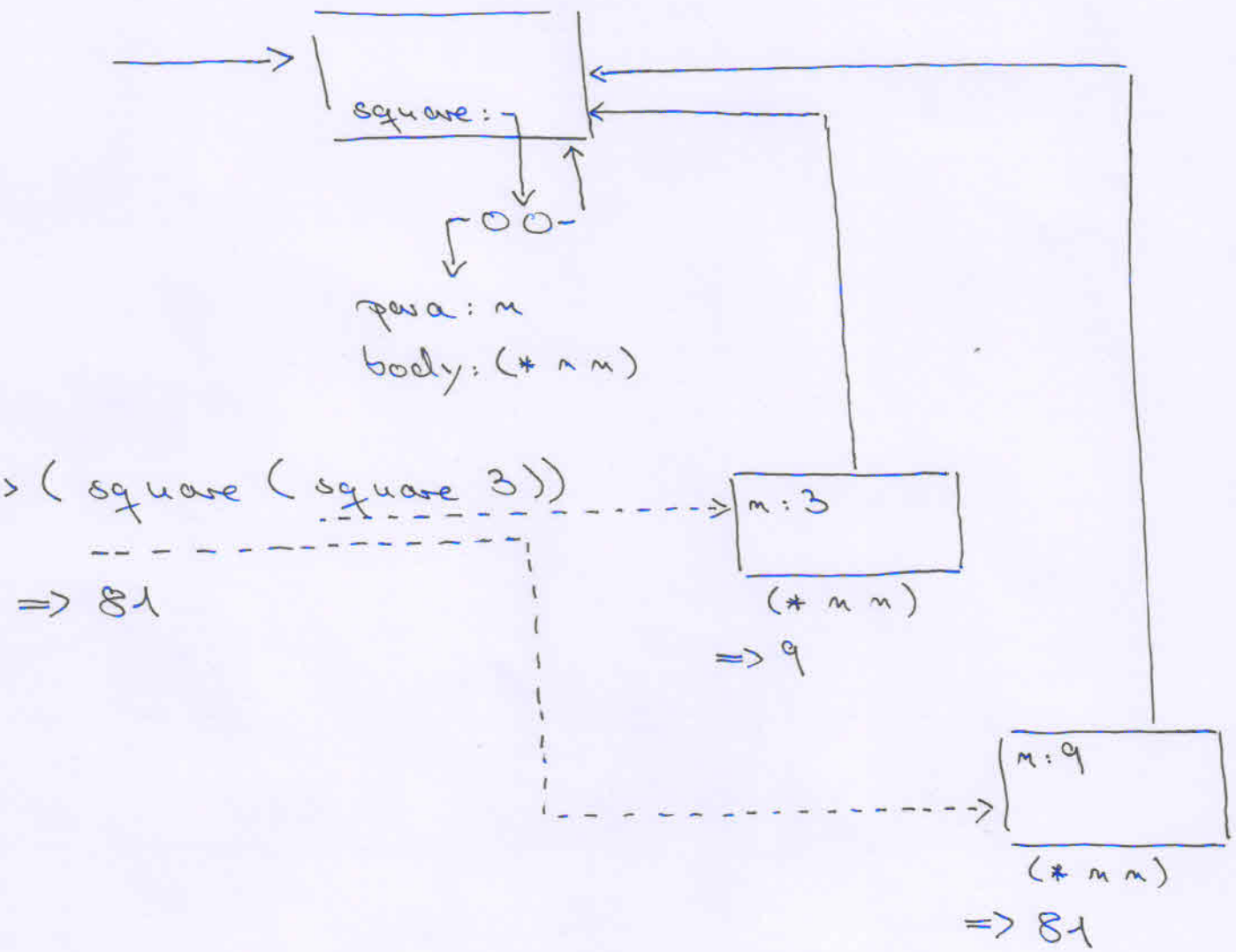
↳ z tym Scheme już nie jest czystym językiem funkcyjnym, ale hybrydowym!

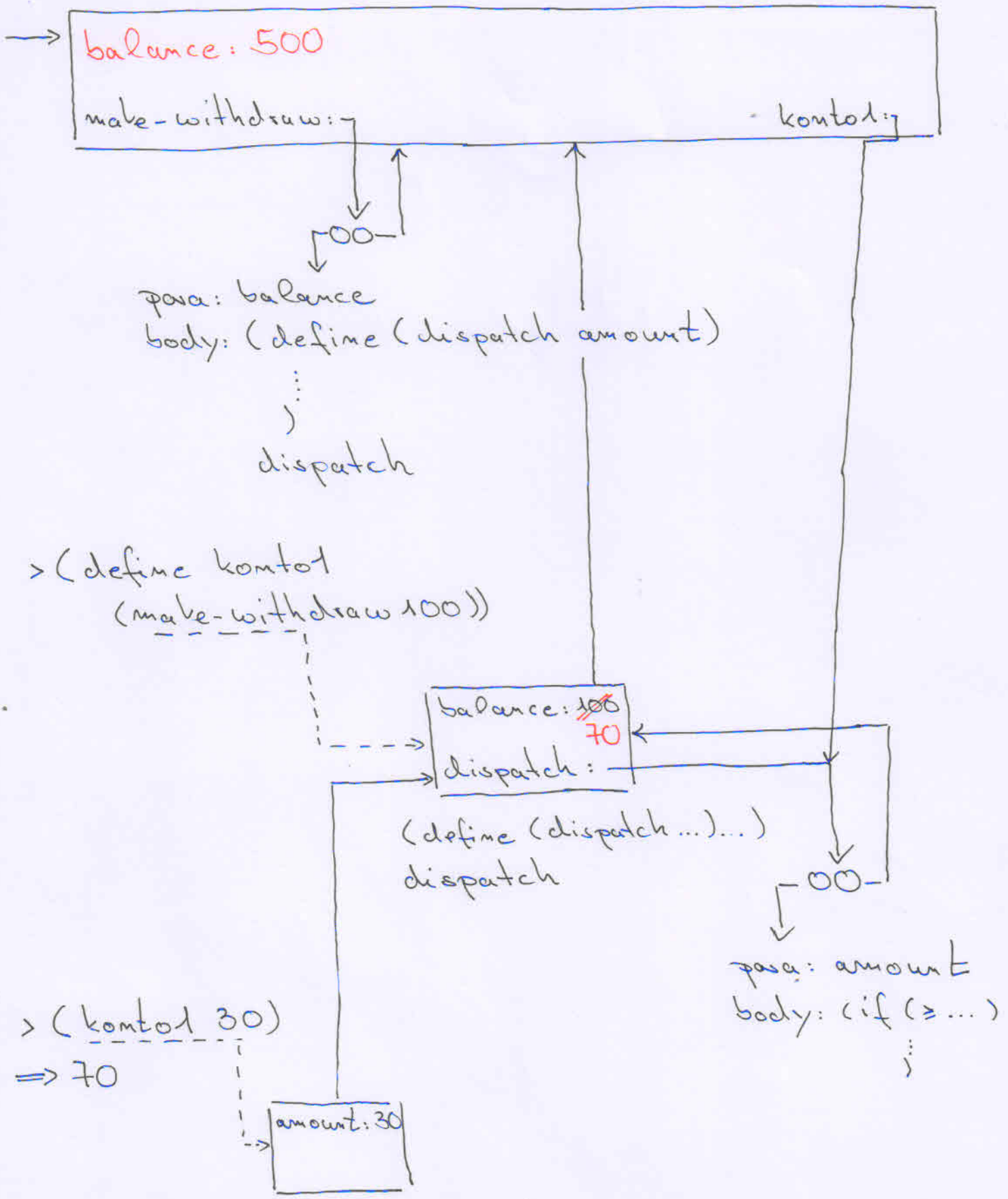
```
(define balance 100)
(define (withdraw amount)
  (if (≥ balance amount)
      (begin (set! balance (- balance amount))
             balance)
      (error "...")))

(define (make-withdraw balance)
  (define (dispatch amount)
    (if (≥ balance amount)
        (begin (set! balance (- balance amount))
               balance)
        (error "...")))
  dispatch)
```

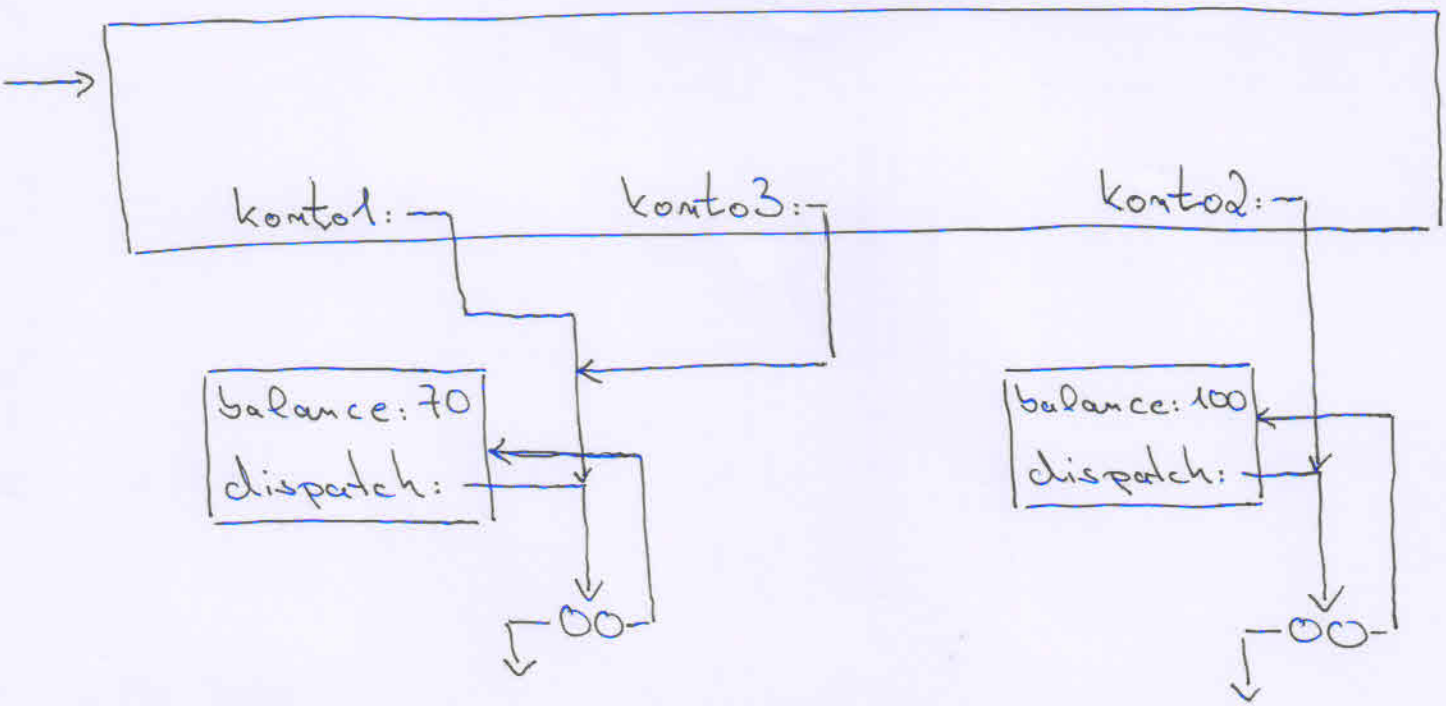
5.1 Rozszerzenie modelu ewaluacji (do modelu środowisk)

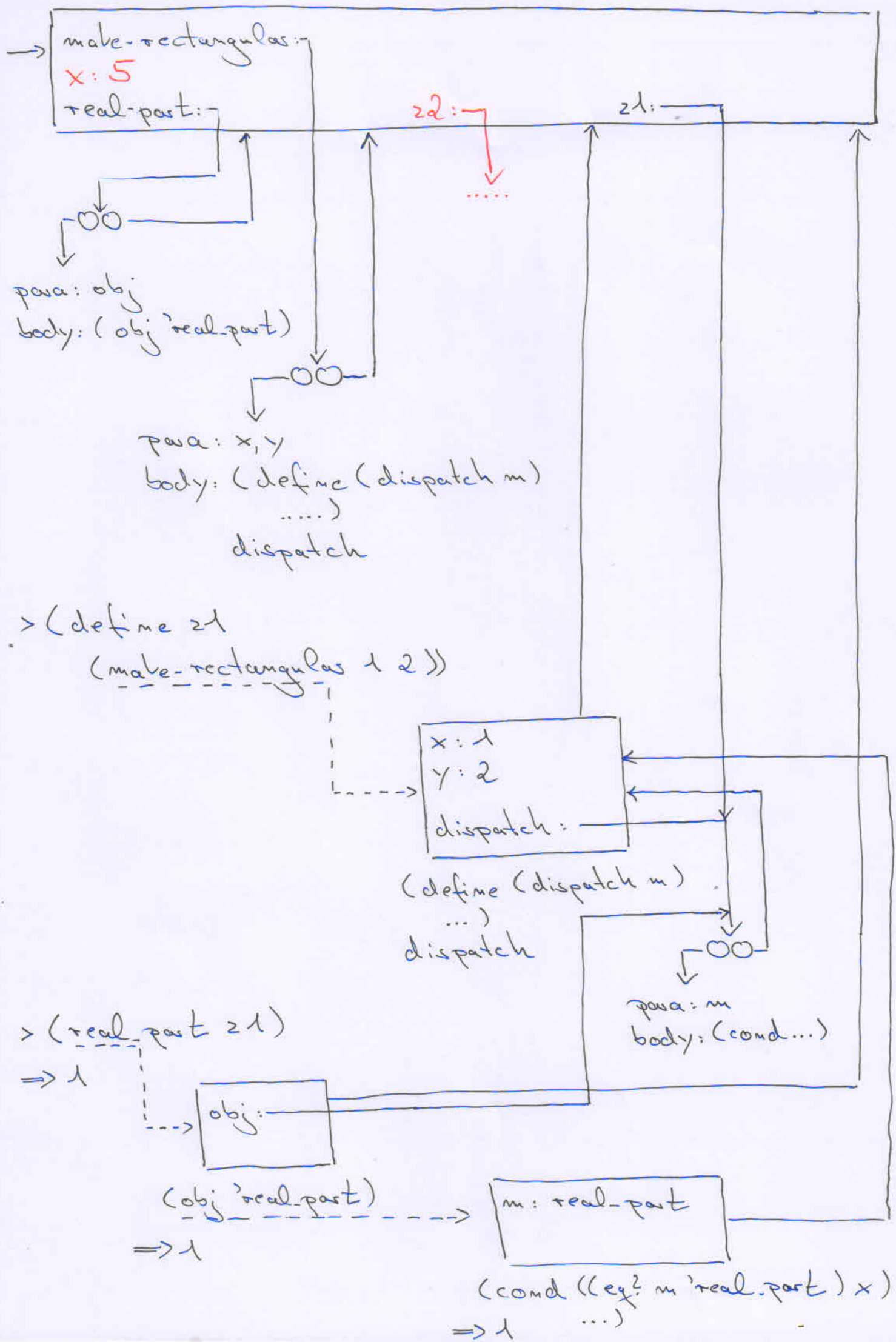
Żeby ewaluować funkcję f, zostaje stworzone nowe środowisko, w którym wartościami formalnych parametrów funkcji f są wartościami aktualnych parametrów. Następca nowego środowiska jest środowiskiem, w którym f została zdefiniowana. W tym nowym środowisku ciało funkcji f zostaje ewaluowane.

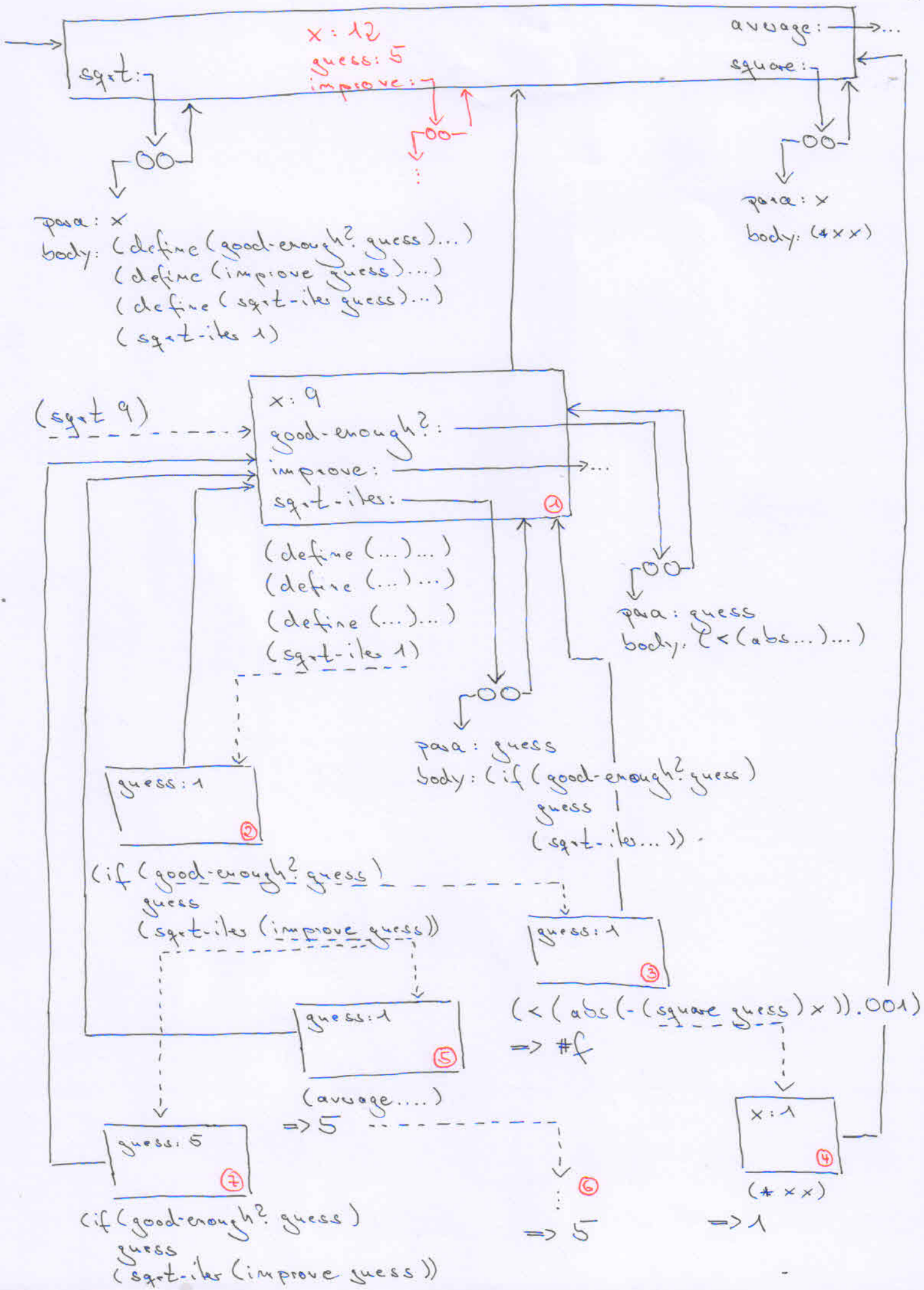



balance: 500
make-withdraw:
konto1:
para: balance
body: (define (dispatch amount)
)
dispatch
> (define konto1
(make-withdraw 100))
balance: 100
70
dispatch:
(define (dispatch ...)...)
dispatch
para: amount
body: (if (≥ ...
)
> (konto1 30)
=> 70
amount: 30
(if (≥ balance amount)
(begin (set! balance...)
...))
=> 70

```
> (define konto2 (make-withdraw 100))
> (define konto3 konto1)
```




make-rectangulas:-
x: 5
real-part:-
z2:-
z1:-
para: obj
body: (obj 'real-part)
para: x, y
body: (define (dispatch m)
....)
dispatch
> (define z1
(make-rectangulas 1 2))
x: 1
y: 2
dispatch:-
(define (dispatch m)
...)
dispatch
para: m
body: (cond...)
> (real-part z1)
=> 1
obj:-
(obj 'real-part)
=> 1
m: real-part
(cond ((eq? m 'real-part) x)
...)
=> 1


x : 12
guess: 5
improve:
sqrt:
average:
square:
para: x
body: (define (good-enough? guess)...)
(define (improve guess)...)
(define (sqrt-iter guess)...)
(sqrt-iter 1)
para: x
body: (* x x)
(sqrt 9)
x: 9
good-enough?:
improve:
sqrt-iter:
1
(define (...)...)
(define (...)...)
(define (...)...)
(sqrt-iter 1)
para: guess
body: (< (abs...)...)
para: guess
body: (if (good-enough? guess)
guess
(sqrt-iter...))
guess: 1
2
(if (good-enough? guess)
guess
(sqrt-iter (improve guess))
guess: 1
3
(< (abs (- (square guess) x)) .001)
=> #f
guess: 1
5
(average....)
=> 5
x: 1
4
(* x x)
=> 1
guess: 5
7
(if (good-enough? guess)
guess
(sqrt-iter (improve guess))
6
=> 5

5.2 Znowó o listach

(set-car! x w) (set-cdr! x w)

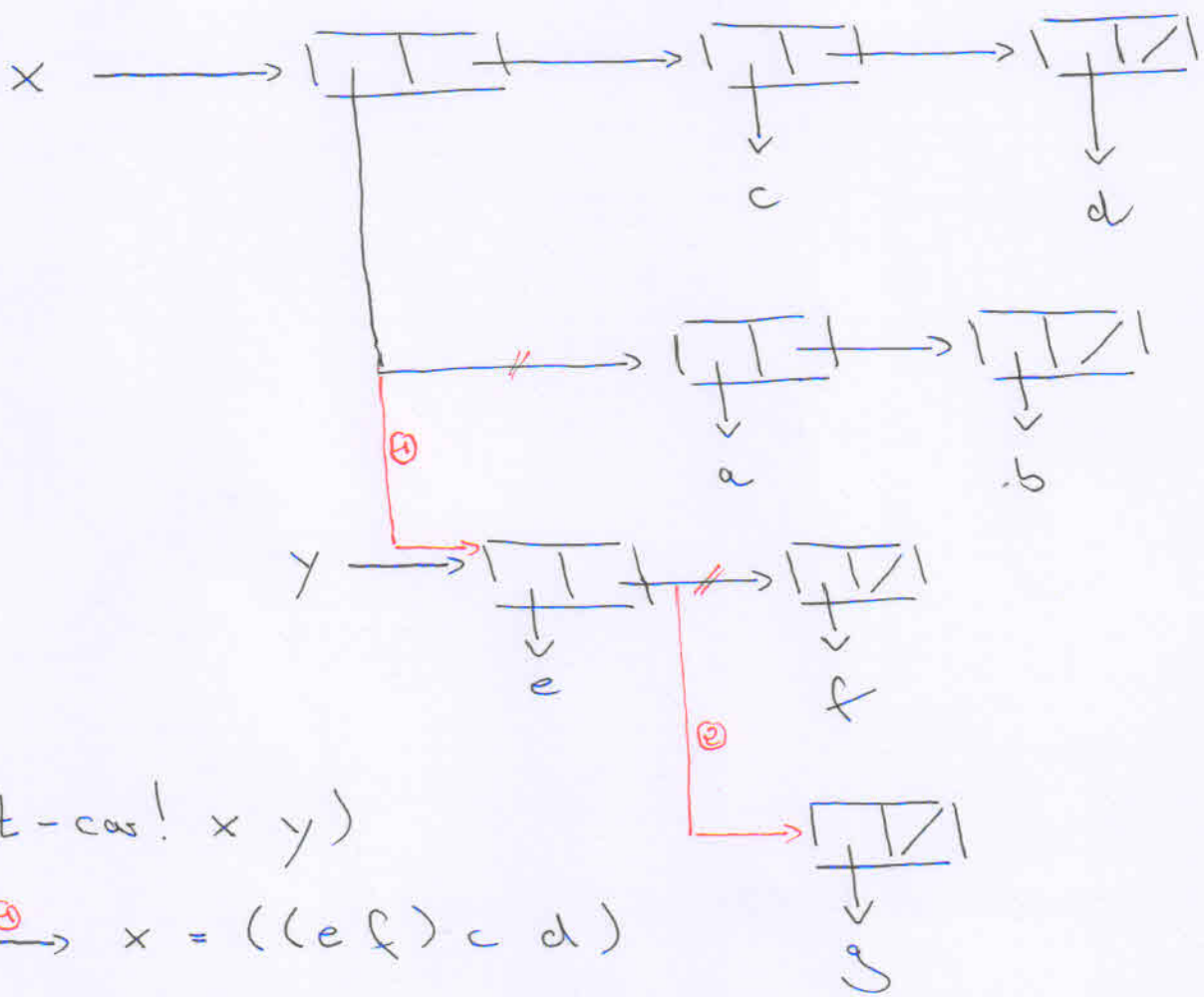
lub

(set-mcar! x w) (set-mcdr! x w)

tylko dla list budowanych z mcons, mcar, mcdr !

Przykład:

```
(define x '((a b) c d))
(define y '(e f))
```



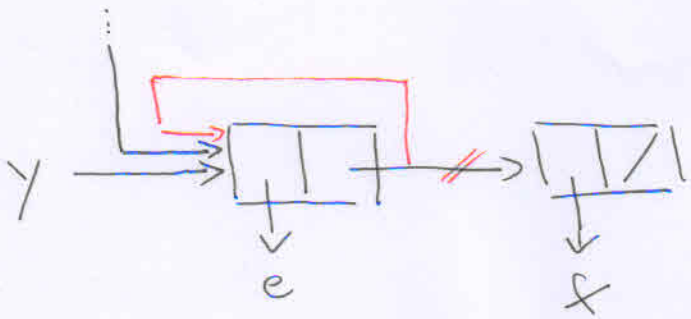
(set-car! x y)

① → x = ((e f) c d)

(set-cdr! y '(g))

② → y = (e g), ale też x = ((e g) c d)

(set-cdr! y y)



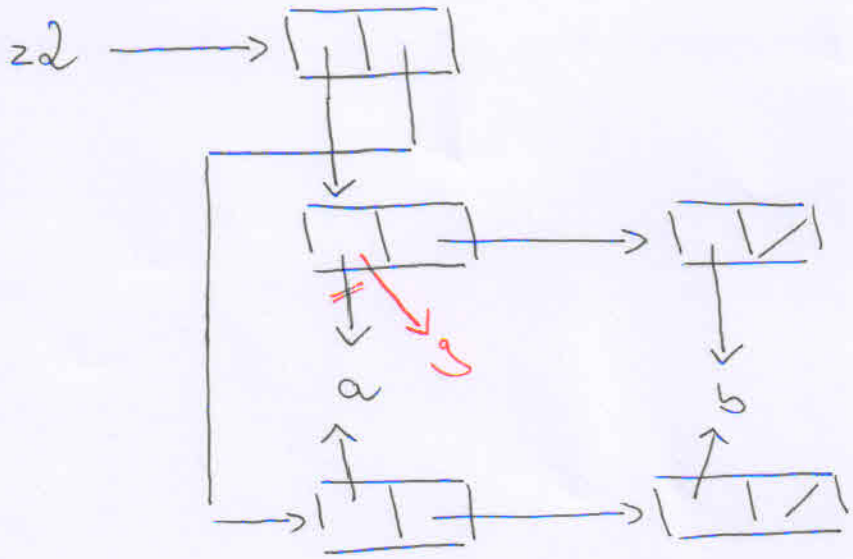
> (length x)
3
> (length y)
błąd

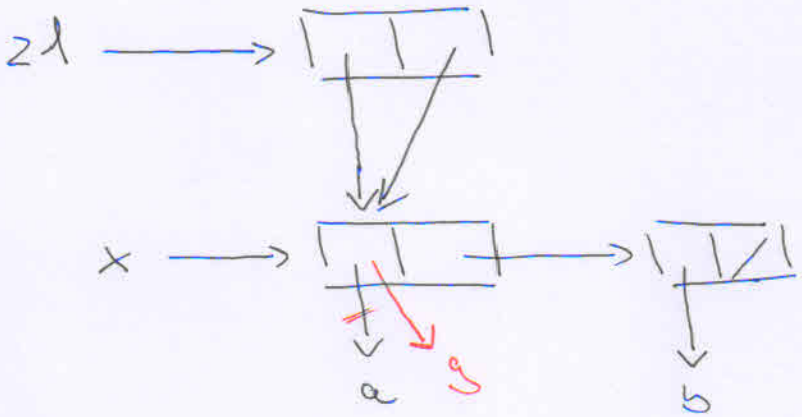
Przykład:

(define x '(a b))
(define z1 (cons x x))
(define z2 (cons '(a b) '(a b)))

> z1
((a b) a b)
> z2
((a b) a b)

> (set-car! (car z1) 'g)
> (set-car! (car z2) 'g)





czyli

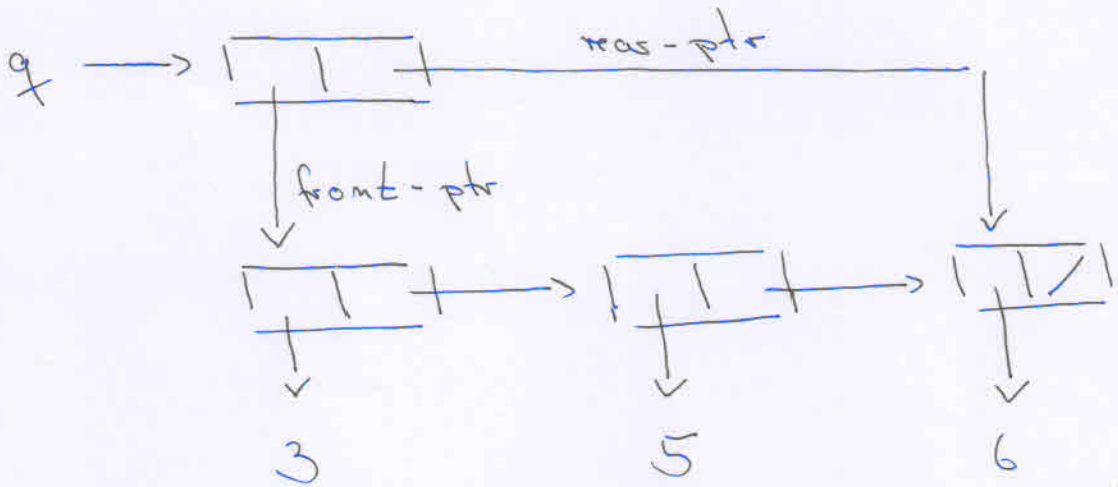
z1 = ((g b) g b)

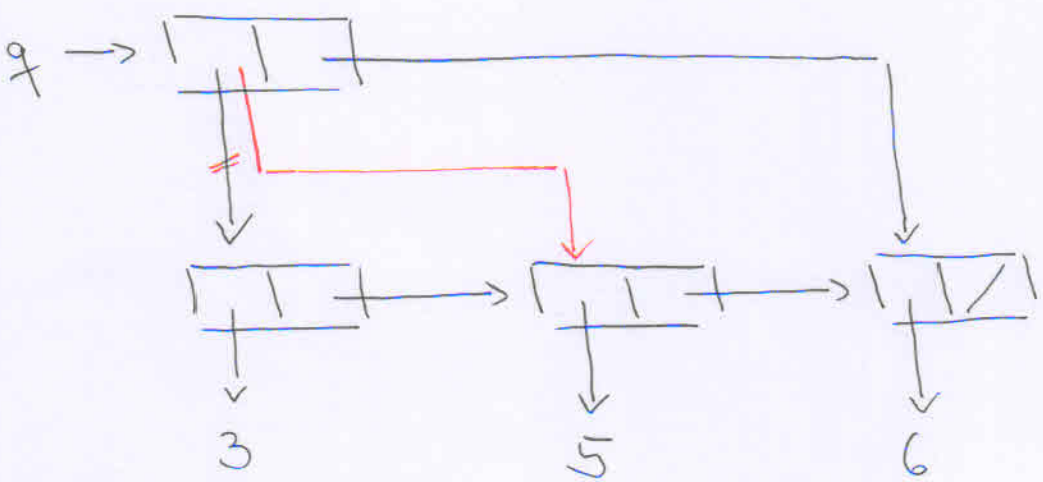
z2 = ((g b) a b)

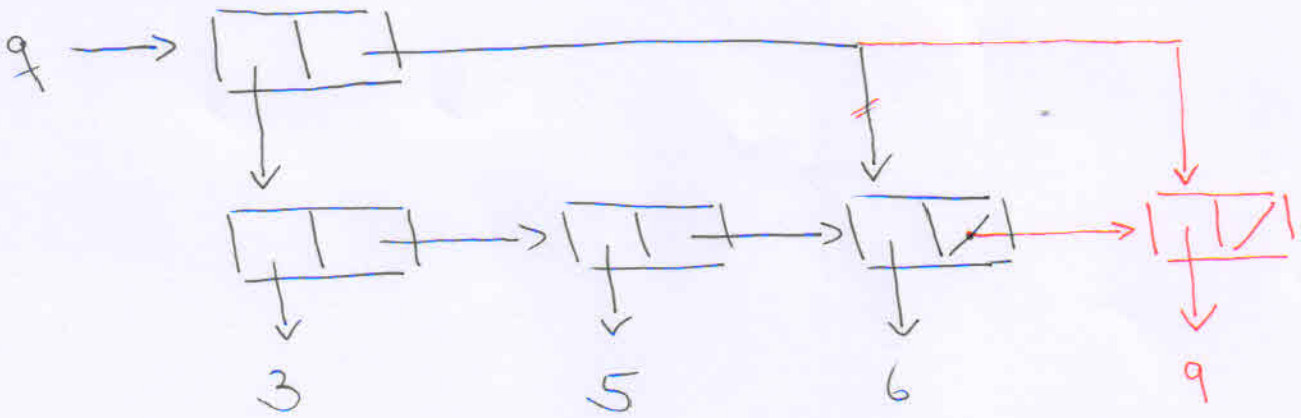
5.3 Kolejki

q = (~~3~~ ~~5~~ 6 9)
↑ ↳ 1 7 ...

(make-queue)
(empty-queue? q)
(front q)
(insert-queue! q i)
(delete-queue! q)








```
(define (make-queue) (mcons '() '()))
(define (front-ptr queue) (mcar queue))
(define (set-front-ptr! queue item)
  (set-mcar! queue item))
(define (empty-queue? queue)
  (null? (front-ptr queue)))
```

13

```
> (define q (make-queue))
> (insert-queue! q 'a)
((a) a)
> (insert-queue! q 'b)
((a b) b)
> (delete-queue! q)
((b) b)
> (delete-queue! q)
(() b)
```

```
;;; Queues

#lang racket

(define (front-ptr queue) (mcar queue))

(define (rear-ptr queue) (mcdr queue))

(define (set-front-ptr! queue item) (set-mcar! queue item))

(define (set-rear-ptr! queue item) (set-mcdr! queue item))

(define (empty-queue? queue) (null? (front-ptr queue)))


(define (make-queue) (mcons '() '()))

(define (front-queue queue)
  (if (empty-queue?)
      (error "Front-queue with empty queue" queue)
      (mcar (front-ptr queue))))

(define (insert-queue! queue item)
  (let ((new (mcons item '())))
    (if (empty-queue? queue)
        (begin (set-front-ptr! queue new)
               (set-rear-ptr! queue new)
               queue)
        (begin (set-mcdr! (rear-ptr queue) new)
               (set-rear-ptr! queue new)
               queue))))

(define (delete-queue! queue)
  (if (empty-queue? queue)
      (error "Delete-queue with empty queue" queue)
      (begin (set-front-ptr! queue (mcdr (front-ptr queue)))
             queue)))

(define (print-queue queue)
   (define (mlist-to-list l)
      (if (null? l)
          l
          (cons (mcar l) (mlist-to-list (mcdr l)))))
  (mlist-to-list (mcar queue)))
```