Les 10: modeleren met muteerbare data

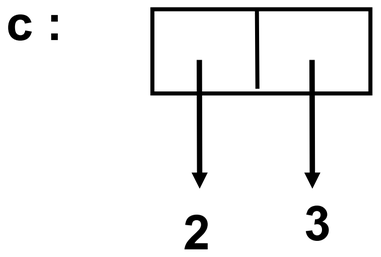
Sessie 1

Les 10: modeleren met muteerbare data

In deze les worden destructieve operatoren geïntroduceerd. We hebben er eigenlijk niet bij stil gestaan maar geen enkele van de operatoren die we tot nu toe gebruikten hadden als effect of neveneffect dat een bestaande structuur werd aangetast. Dat wordt vanaf nu anders. Het is belangrijk om in te zien dat destructieve operatoren zorgvuldig moeten gebruikt worden en dat zij programmeren voor een stuk gevaarlijker en dus moeilijker maken. Ook de problematiek van 'gelijkheid' moet opnieuw worden bekeken.

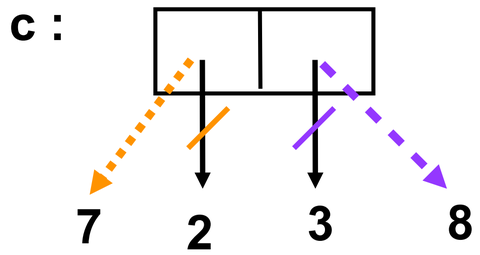
Muteerbare lijststructuren - basisoperatoren

```
> (define c (cons 2 3))
> c
(2 . 3)
> (set-car! c 7 )
> c
(7 . 3)
> (set-cdr! c 8 )
> c
(7 . 8)
```



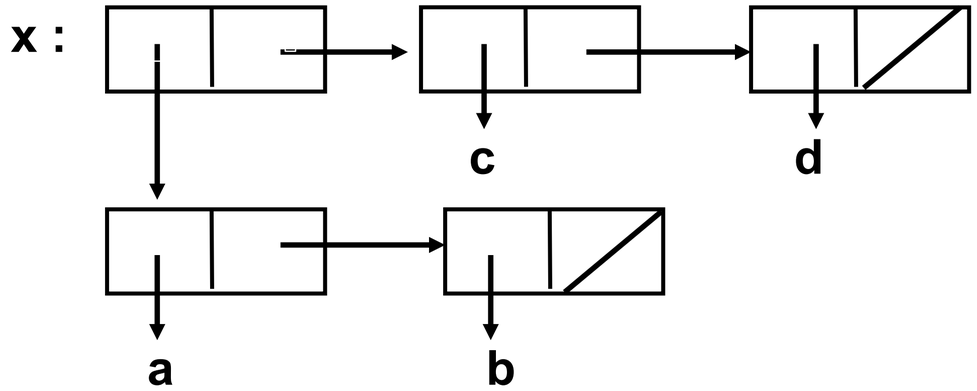
Muteerbare lijststructuren - basisoperatoren

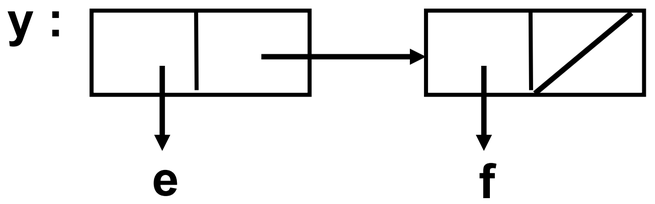
```
> (define c (cons 2 3))
> c
(2 . 3)
> (set-car! c 7 )
> c
(7 . 3)
> (set-cdr! c 8 )
> c
(7 . 8)
```

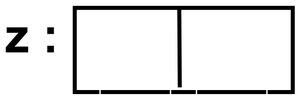


Muteerbare lijststructuren - voorbeeld

```
> (define x '((a b) c d))
> (define y '(e f))
> (define z (cons y (cdr x)))
> x
((a b) c d)
> y
(e f)
> z
((e f) c d)
> (set-car! x y)
> x
((e f) c d)
> y
(e f)
> (set-cdr! x y)
> x
((e f) e f)
```

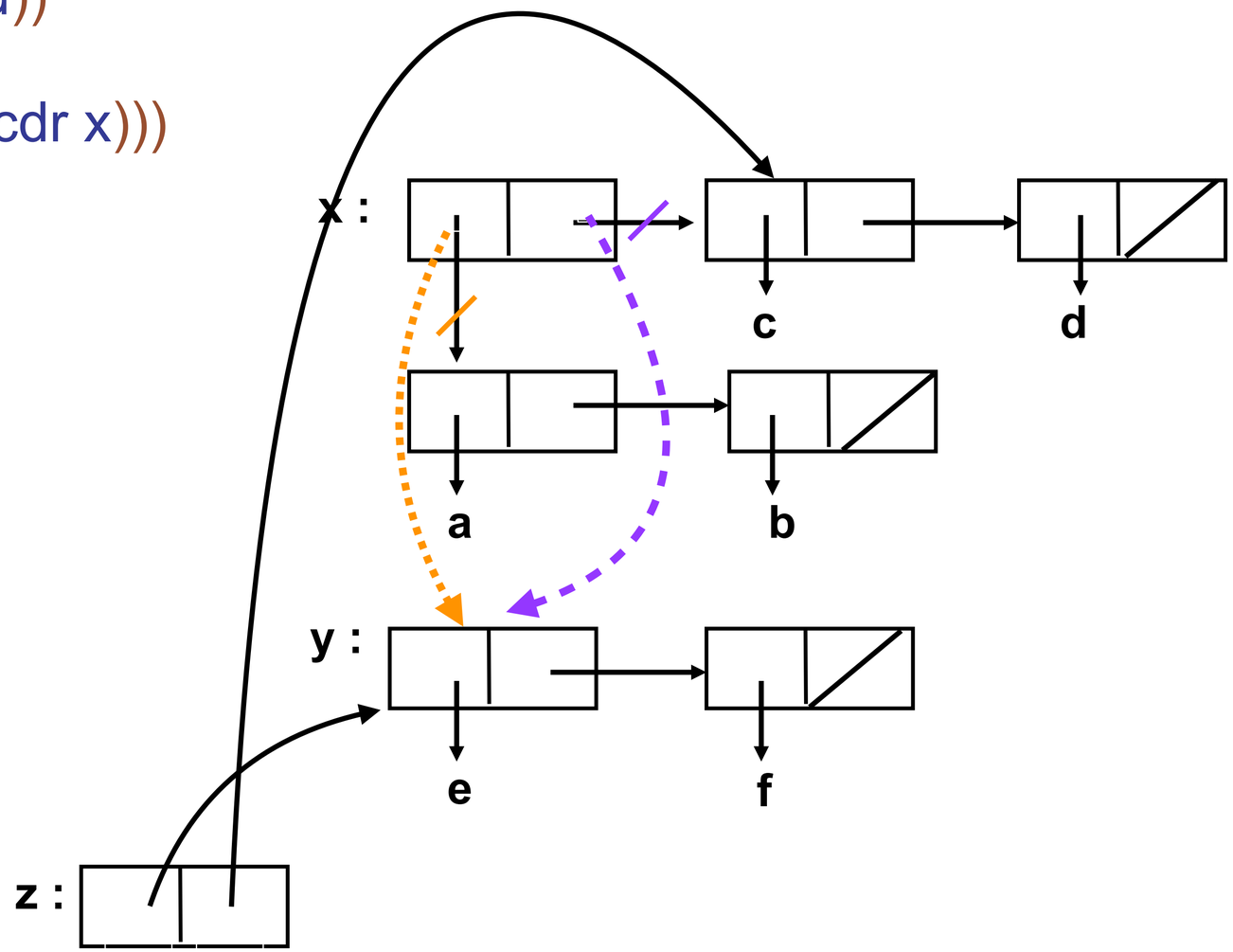






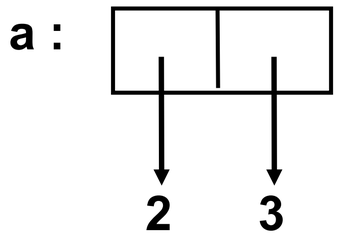
Muteerbare lijststructuren - voorbeeld

```
> (define x '((a b) c d))
> (define y '(e f))
> (define z (cons y (cdr x)))
> x
((a b) c d)
> y
(e f)
> z
((e f) c d)
> (set-car! x y)
> x
((e f) c d)
> y
(e f)
> (set-cdr! x y)
> x
((e f) e f)
```

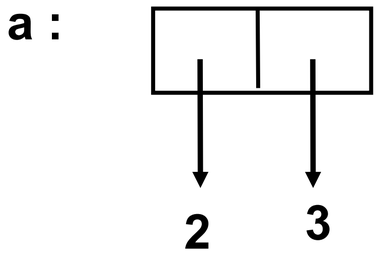


Gelijkheid en verandering - nieuwe cons cel <> cons cel aanpassen

```
> (define a (cons 2 3))
> a
(2 . 3)
> (set! a (cons 1 (cdr a)))
> a
(1 . 3)
```



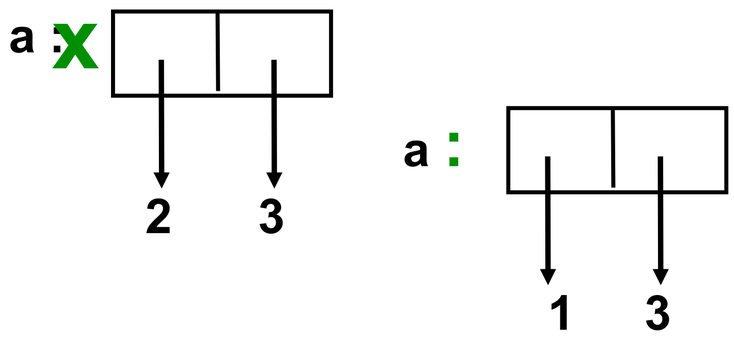

```
> (define a (cons 2 3))
> a
(2 . 3)
> (set-car! a 1)
> a
(1 . 3)
```

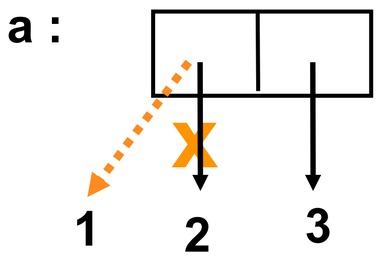


Gelijkheid en verandering - nieuwe cons cel <> cons cel aanpassen

```
> (define a (cons 2 3))
> a
(2 . 3)
> (set! a (cons 1 (cdr a)))
> a
(1 . 3)
```

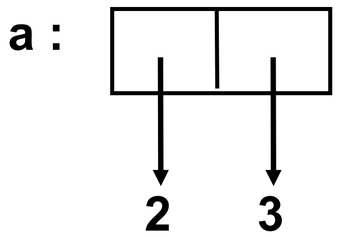
```
> (define a (cons 2 3))
> a
(2 . 3)
> (set-car! a 1)
> a
(1 . 3)
```



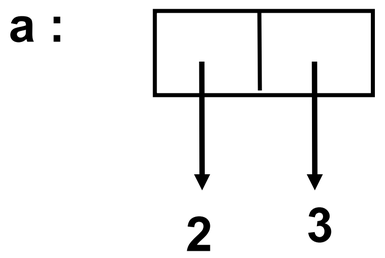


Gelijkheid en verandering - aliases

```
> (define a (cons 2 3))
> (define b a)
> a
(2 . 3)
> b
(2 . 3)
> (set! a (cons 1 (cdr a)))
> a
(1 . 3)
> b
(2 . 3)
```

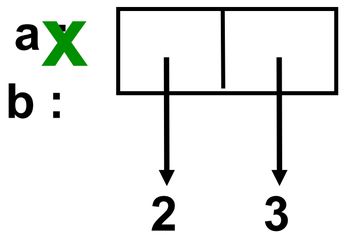


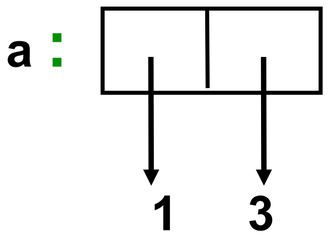

```
> (define a (cons 2 3))
> (define b a)
> a
(2 . 3)
> b
(2 . 3)
> (set-car! a 1)
> a
(1 . 3)
> b
(1 . 3)
```



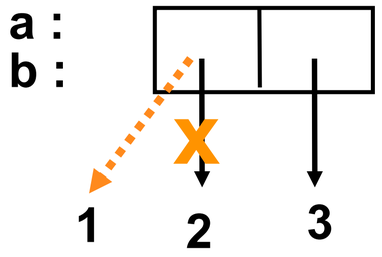
Gelijkheid en verandering - aliases

```
> (define a (cons 2 3))
> (define b a)
> a
(2 . 3)
> b
(2 . 3)
> (set! a (cons 1 (cdr a)))
> a
(1 . 3)
> b
(2 . 3)
```





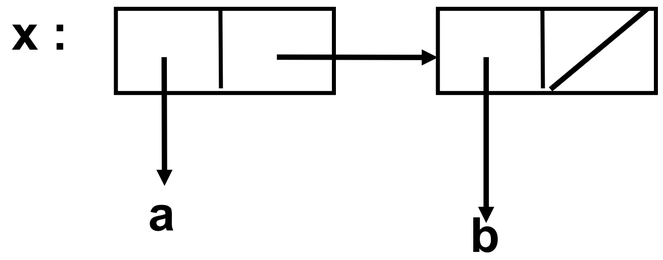

```
> (define a (cons 2 3))
> (define b a)
> a
(2 . 3)
> b
(2 . 3)
> (set-car! a 1)
> a
(1 . 3)
> b
(1 . 3)
```

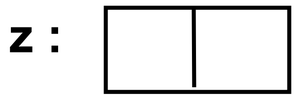


Delen en identiteit (1)

```
> (define x (list 'a 'b))
> x
(a b)
> (define z (cons x x))
> z
((a b) a b)
> (set-car! x 'c)
> x
(c b)
> z
((c b) c b)
> (set-car! (cdar z) 'd)
> z
((c d) c d)
```

zowel de car als de cdr van z wijzen naar dezelfde lijst





let op de 2 c's

je kan ook navigeren naar een cons-cel en dan een verandering aanbrengen

en de 2 d's

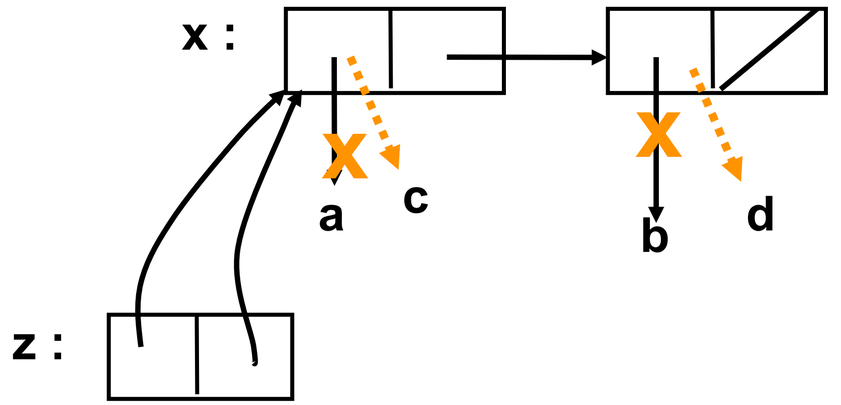
Delen en identiteit (1)

```
> (define x (list 'a 'b))
> x
(a b)
> (define z (cons x x))
> z
((a b) a b)
> (set-car! x 'c)
> x
(c b)
> z
((c b) c b)
> (set-car! (cdar z) 'd)
> z
((c d) c d)
```

zowel de car als de cdr van z wijzen naar dezelfde lijst

let op de 2 c's

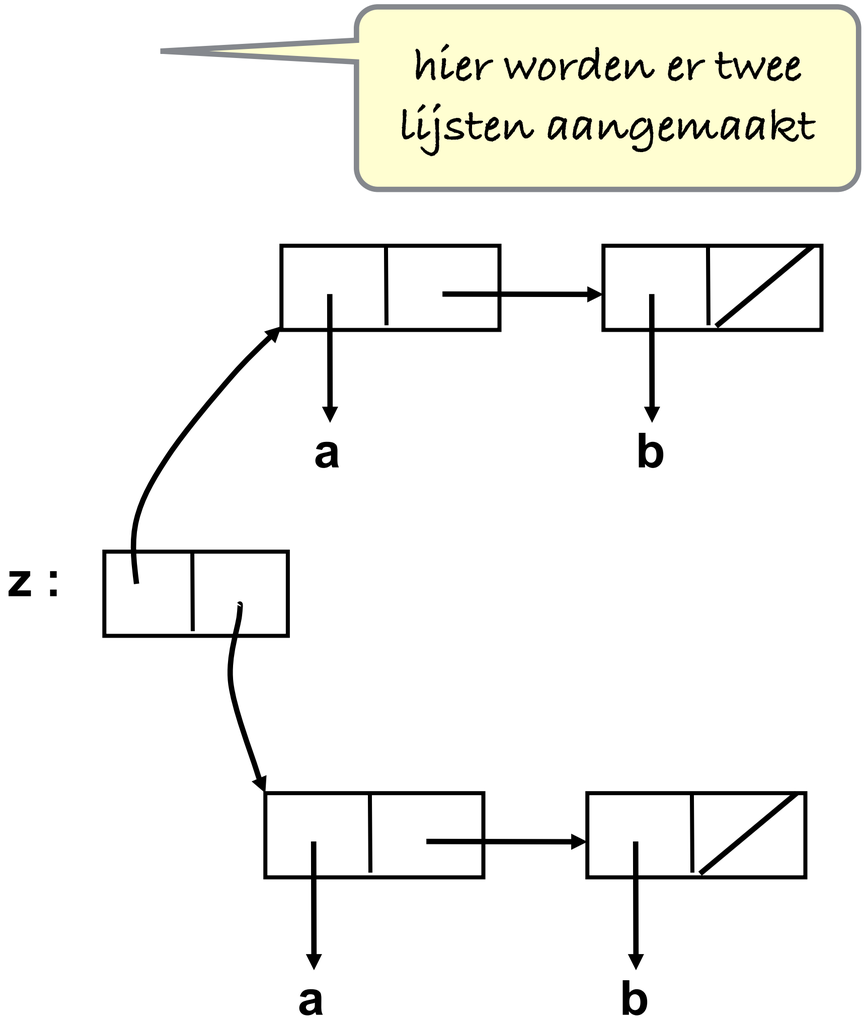
en de 2 d's



je kan ook navigeren naar een cons-cel en dan een verandering aanbrengen

Delen en identiteit (2)

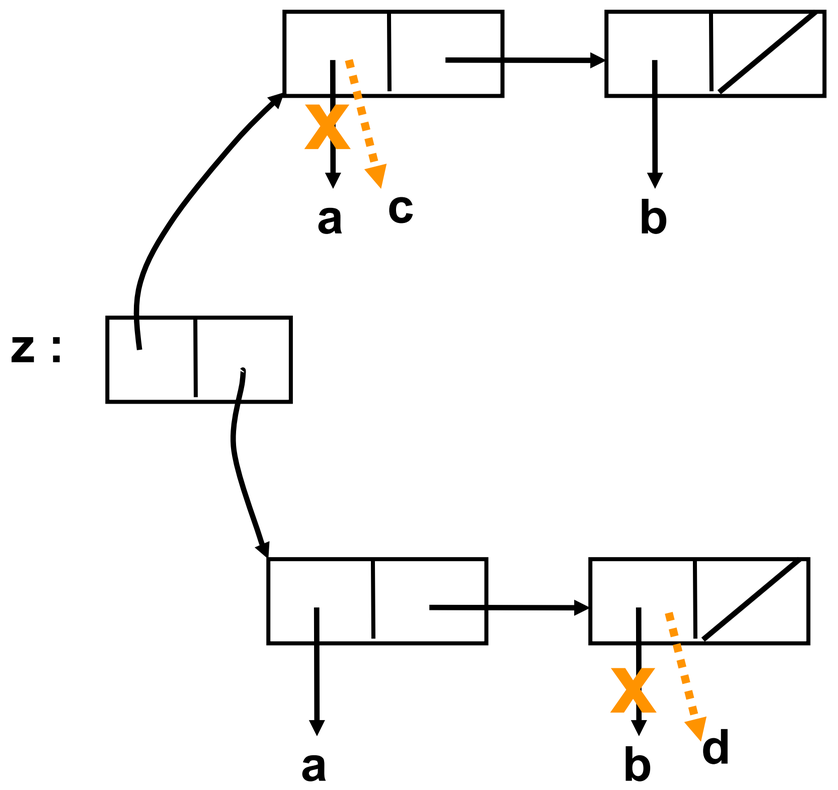
```
> (define z (cons (list 'a 'b) (list 'a 'b)))
> z
((a b) a b)
> (set-car! (car z) 'c)
> z
((c b) a b)
> (set-car! (cddr z) 'd)
> z
((c b) a d)
```



Delen en identiteit (2)

```
> (define z (cons (list 'a 'b) (list 'a 'b)))
> z
((a b) a b)
> (set-car! (car z) 'c)
> z
((c b) a b)
> (set-car! (cddr z) 'd)
> z
((c b) a d)
```

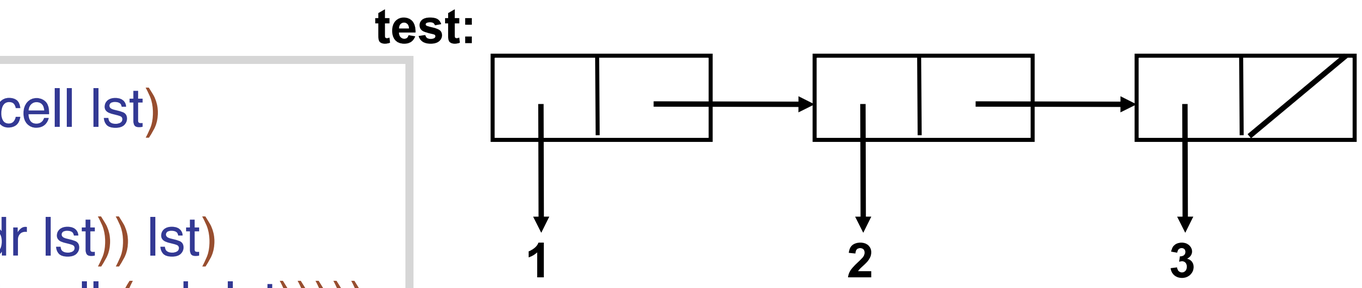
hier worden er twee lijsten aangemaakt



Voorbeeld: circulaire lijst

```
(define (last-cell lst)
  (cond
     ((null? (cdr lst)) lst)
     (else (last-cell (cdr lst)))))

(define (make-cycle lst)
  (set-cdr! (last-cell lst) lst))
```



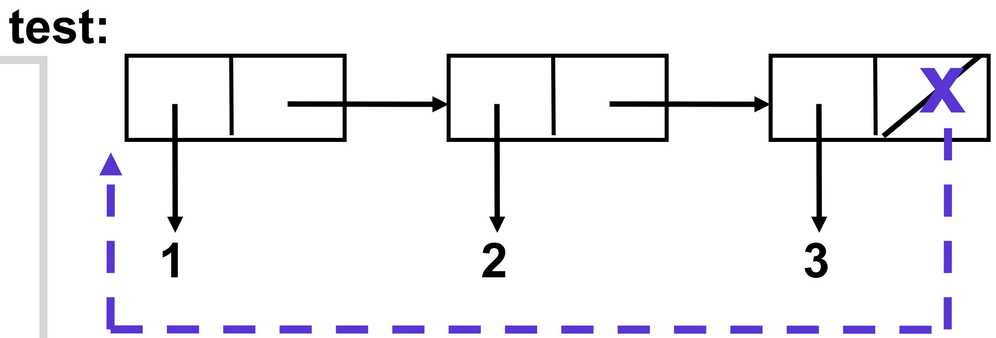

```
(define (nth n lst)
  (if (= n 0)
      (car lst)
      (nth (- n 1) (cdr lst))))
```

Voorbeeld: circulaire lijst

```
(define (last-cell lst)
  (cond
     ((null? (cdr lst)) lst)
     (else (last-cell (cdr lst)))))

(define (make-cycle lst)
  (set-cdr! (last-cell lst) lst))
```

```
(define (nth n lst)
  (if (= n 0)
      (car lst)
      (nth (- n 1) (cdr lst))))
```




```
> (define test '(1 2 3))
> test
(1 2 3)
> (make-cycle test)
> test
#0=(1 2 3 . #0#)
> (cadr test)
2
> (caddr test)
3
> (cadddr test)
1
```

```
> (nth 8 test)
3
> (nth 100 test)
2
```

Les 10: modeleren met muteerbare data

Sessie 2

Voorbeeld: append

classic append

```
(define (append lst1 lst2)
   (if (null? lst1)
       lst2
       (cons (car lst1)
          (append (cdr lst1) lst2))))
```

```
> (define p '(1 2 3))
> (define q '(4 5))
> (append p q)
(1 2 3 4 5)
> p
(1 2 3)
> q
(4 5)
```

de originele lijsten zijn niet veranderd

mutator append

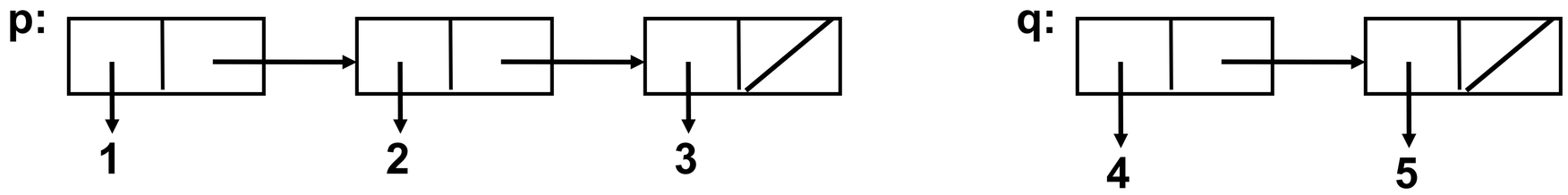
```
(define (append! lst1 lst2)
   (set-cdr! (last-cell lst1) lst2)
   lst1)
```

```
> (define p '(1 2 3))
> (define q '(4 5))
> (append! p q)
(1 2 3 4 5)
> p
(1 2 3 4 5)
> q
(4 5)
```

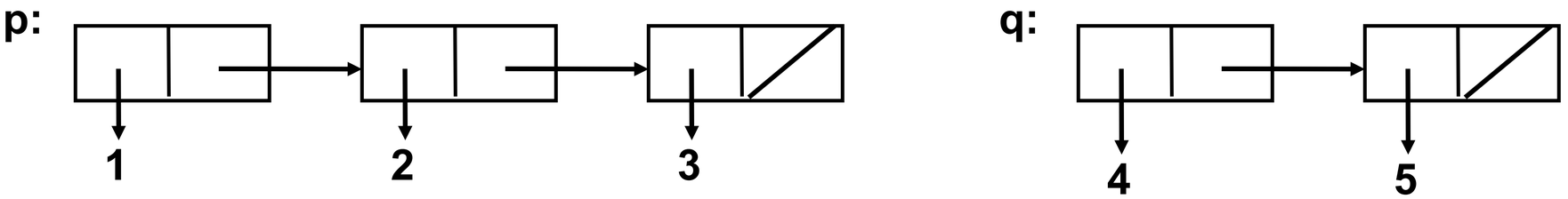
de eerste lijst is aangetast

Klassieke versus mutator append

(append p q)

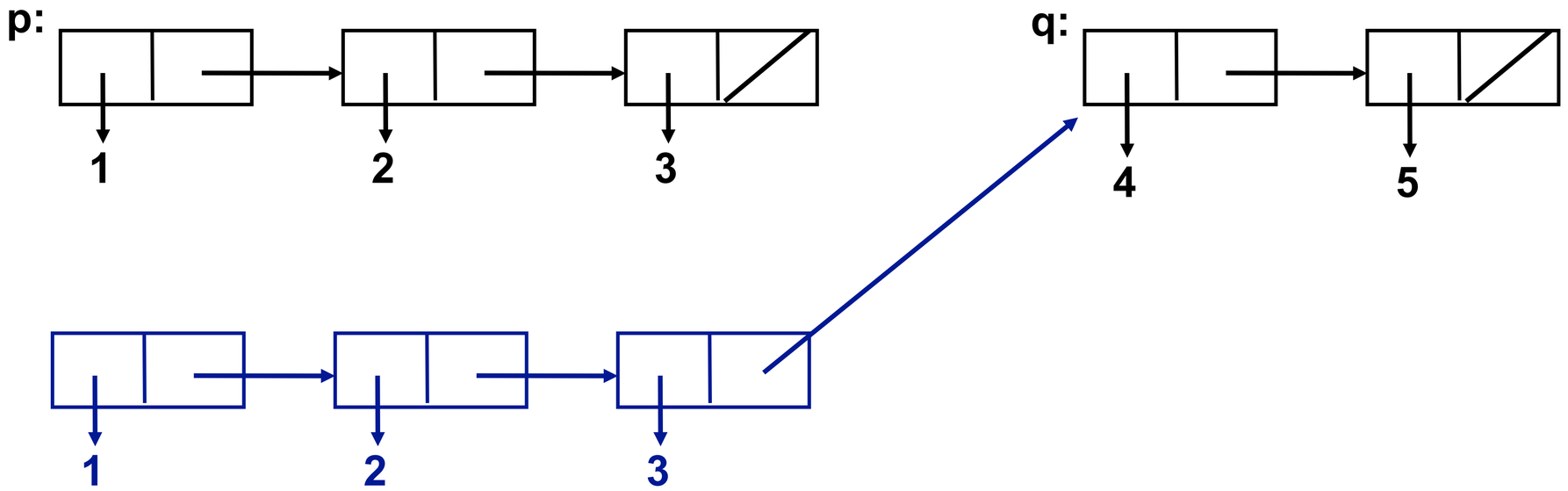


(append! p q)

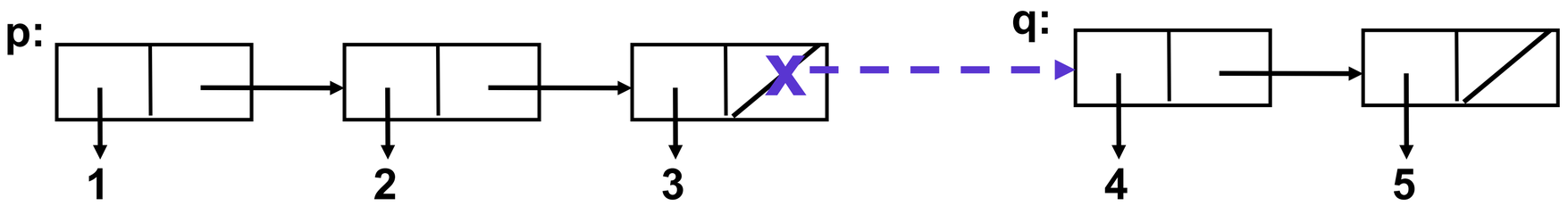


Klassieke versus mutator append

(append p q)



(append! p q)



Voorbeeld: replace

classic replace

```
(define (replace old new lst)
  (cond
  ((null? lst) '())
  ((eq? (car lst) old)
   (cons new
     (replace old new (cdr lst))))
   (else
    (cons (car lst)
      (replace old new (cdr lst))))))
```

```
> (define test '(3 5 2 3 5 3 4))
> (replace 3 9 test)
(9 5 2 9 5 9 4)
> test
(3 5 2 3 5 3 4)
```

de originele lijst is niet veranderd

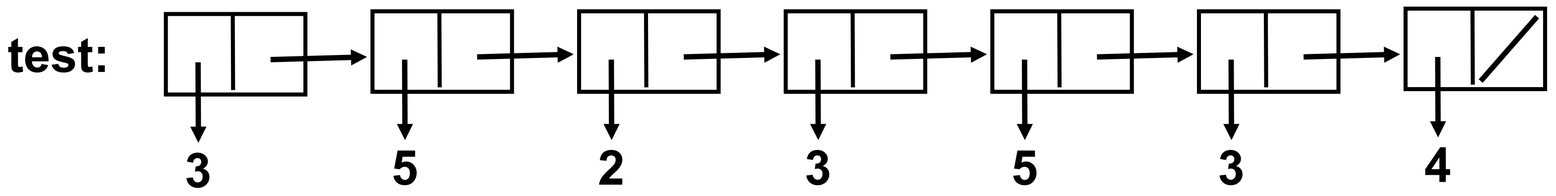
mutator replace

```
(define (replace! old new lst)
  (cond
    ((null? lst) 'done)
    ((eq? (car lst) old)
     (set-car! lst new)
     (replace! old new (cdr lst)))
    (else
     (replace! old new (cdr lst)))))
```

```
> (define test '(3 5 2 3 5 3 4))
> (replace! 3 9 test)
done
> test
(9 5 2 9 5 9 4)
```

de lijst is aangepast

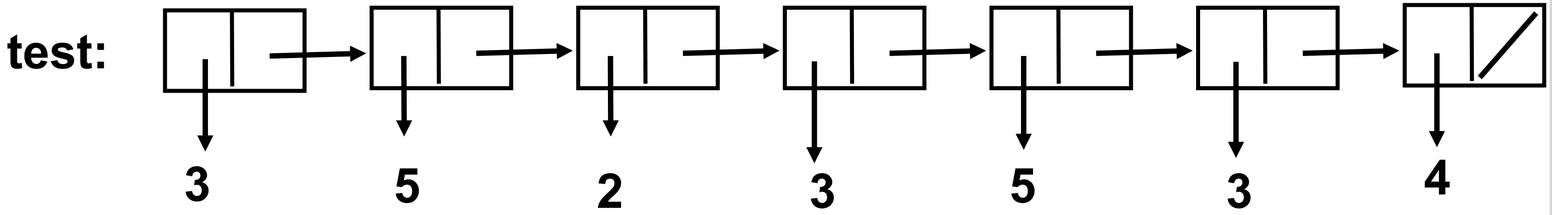
Klassieke versus mutator replace



(replace 3 9 test)

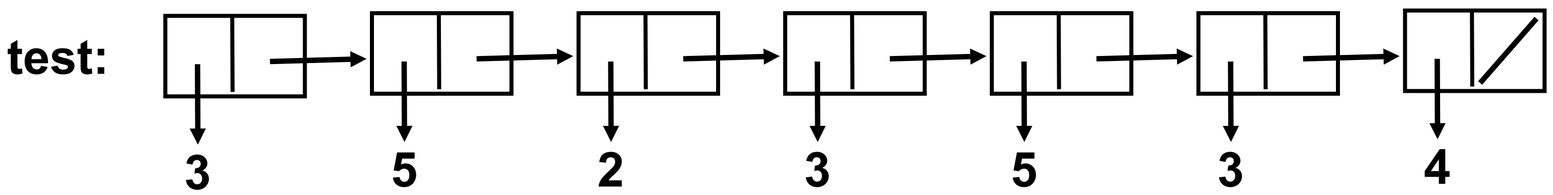
```
(define (replace old new lst)
  (cond
   ((null? lst) '())
   ((eq? (car lst) old)
    (cons new
      (replace old new (cdr lst))))
   (else
    (cons (car lst)
      (replace old new (cdr lst))))))
```

(replace! 3 9 test)

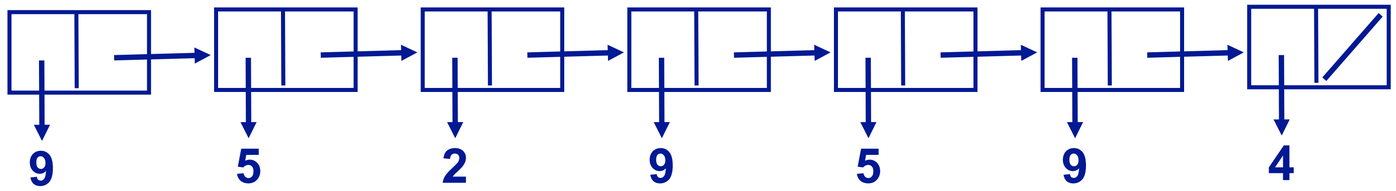



```
(define (replace! old new lst)
  (cond
    ((null? lst) 'done)
    ((eq? (car lst) old)
     (set-car! lst new)
     (replace! old new (cdr lst)))
    (else
     (replace! old new (cdr lst)))))
```

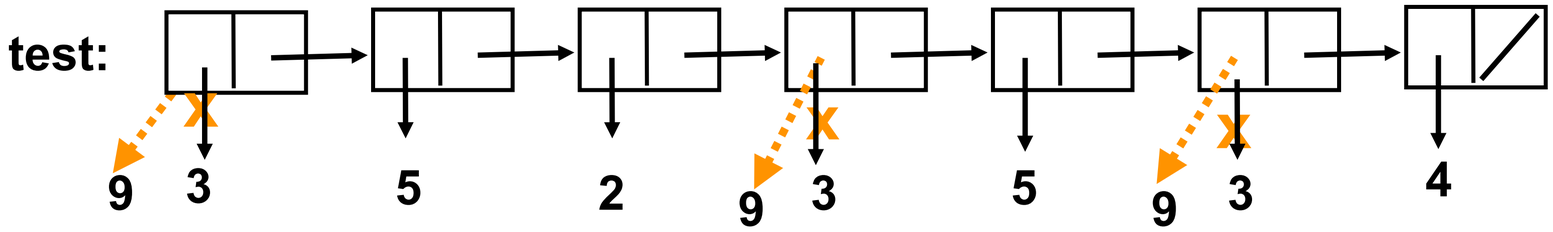
Klassieke versus mutator replace



(replace 3 9 test)



(replace! 3 9 test)



Voorbeeld: remove

the classic remove

```
(define (remove el lst)
  (cond
    ((null? lst) '())
    ((eq? (car lst) el)
     (remove el (cdr lst)))
    (else
     (cons (car lst)
           (remove el (cdr lst))))))
```

```
> (define test '(3 5 2 3 5 3 4))
> (remove 3 test)
(5 2 5 4)
> test
(3 5 2 3 5 3 4)
```

de originele lijst is niet veranderd

the mutator remove

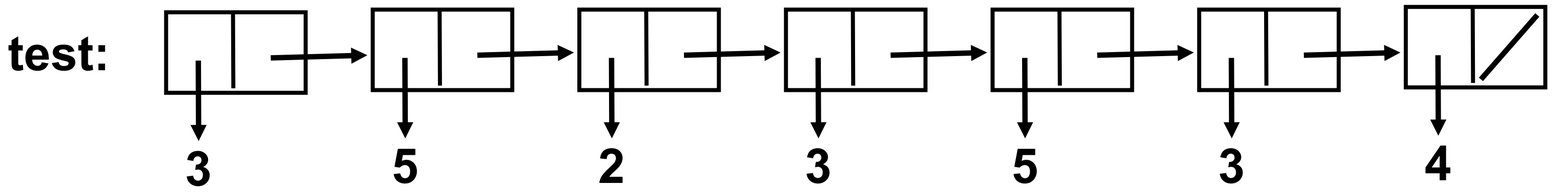
```
(define (remove! el lst)
  (cond
    ((null? lst) 'done)
    ((null? (cdr lst)) 'done)
    ((eq? el (cadr lst))
     (set-cdr! lst (cddr lst))
     (remove! el lst))
    (else
     (remove! el (cdr lst)))))
```

je moet 1 cel vooruit voelen

```
> (define test '(3 5 2 3 5 3 4))
> (remove! 3 test)
done
> test
(3 5 2 5 4)
```

de lijst is niet correct aangepast

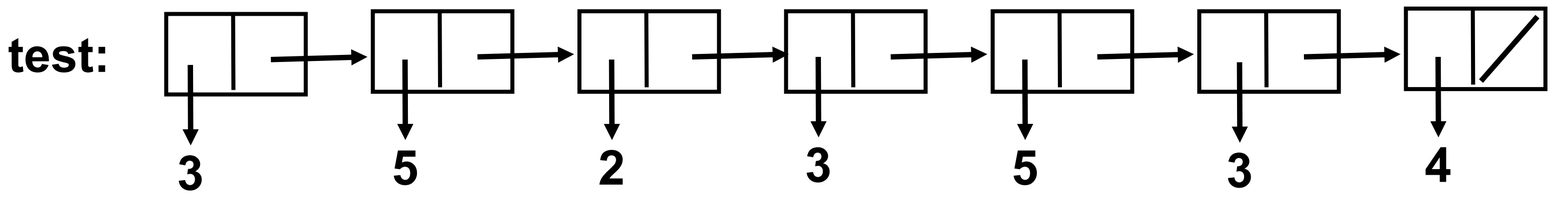
Klassieke versus mutator remove



(remove 3 9 test)

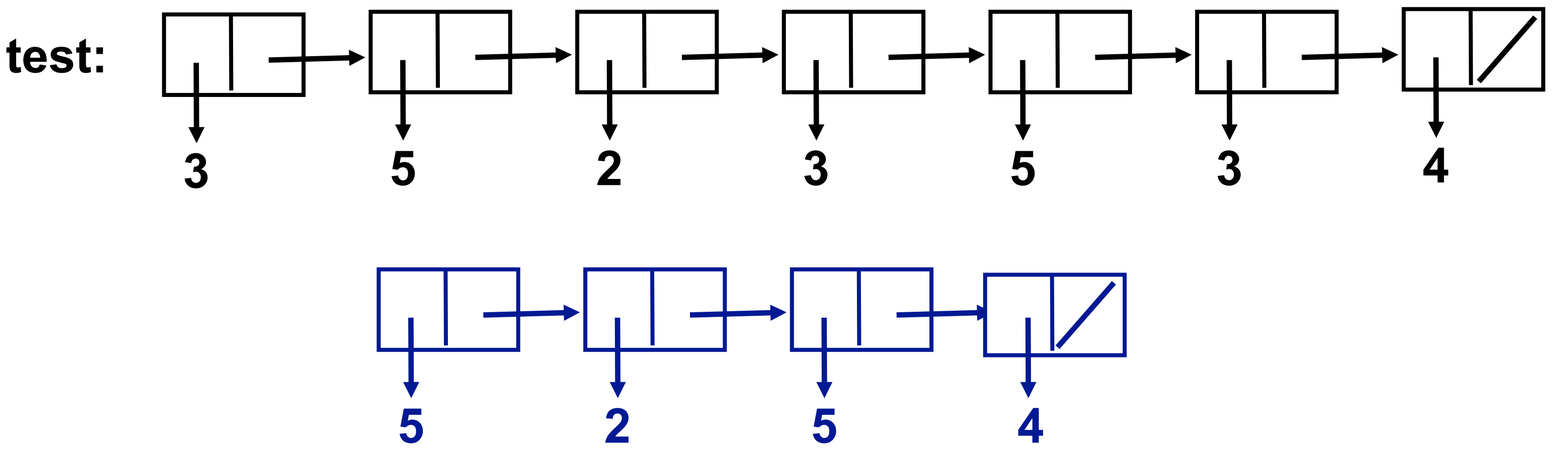
```
(define (remove el lst)
  (cond
    ((null? lst) '())
    ((eq? (car lst) el)
     (remove el (cdr lst)))
    (else
     (cons (car lst)
           (remove el (cdr lst))))))
```

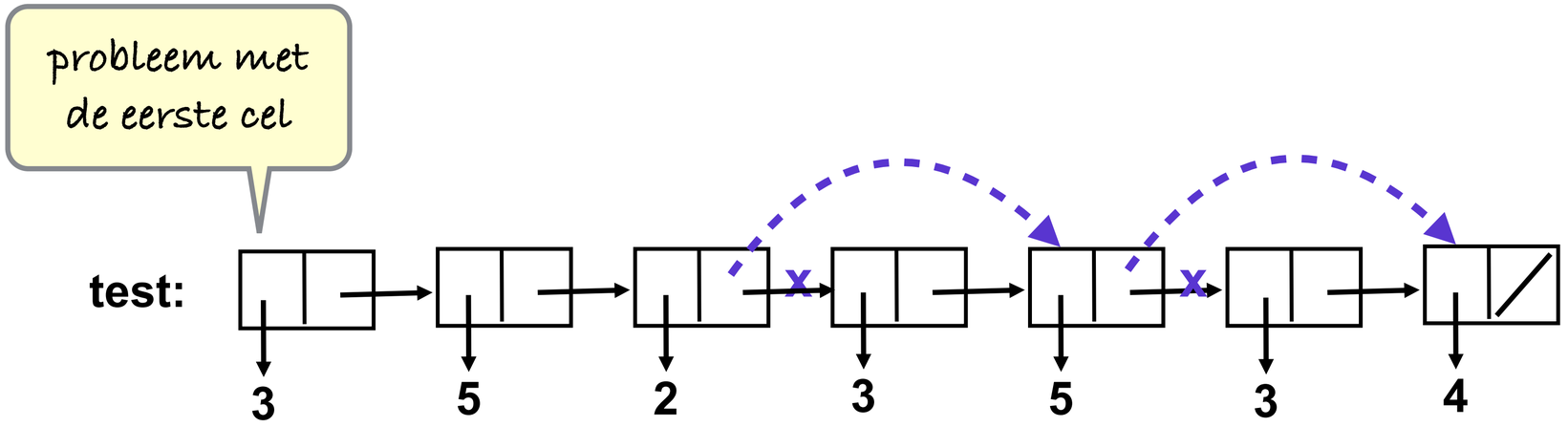
(remove! 3 9 test)




```
(define (remove! el lst)
  (cond
    ((null? lst) 'done)
    ((null? (cdr lst)) 'done)
    ((eq? el (cadr lst))
     (set-cdr! lst (cddr lst))
     (remove! el lst))
    (else
     (remove! el (cdr lst)))))
```

Klassieke versus mutator remove





Eerste cel levert problemen

probeer om de eerste cel
destructief te verwijderen

```
(define (delete-first! lst)
   (set! lst (cdr lst)))
```

```
> (define test '(1 2 3))
> test
(1 2 3)
> (delete-first! test)
> test
(1 2 3)
```

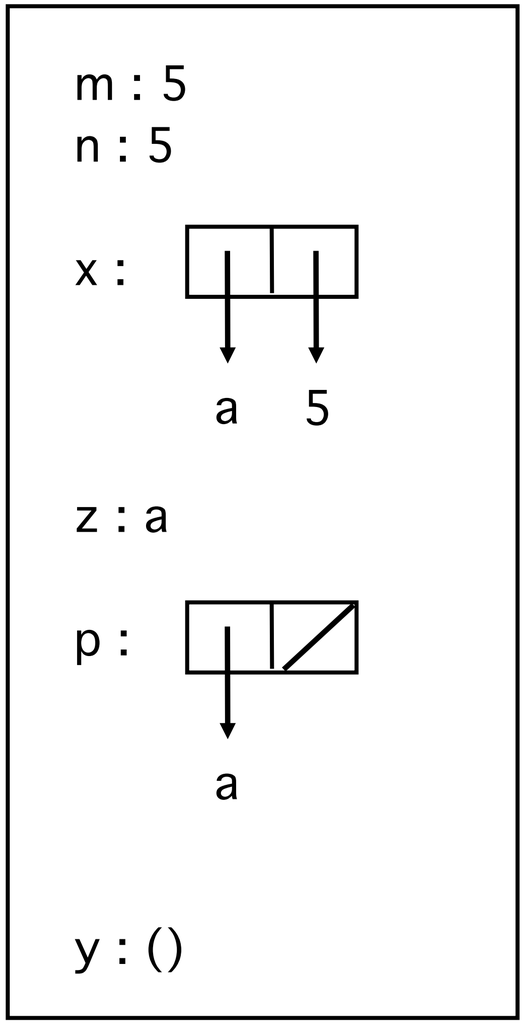
probeer om een cel vooraan
destructief toe te voegen

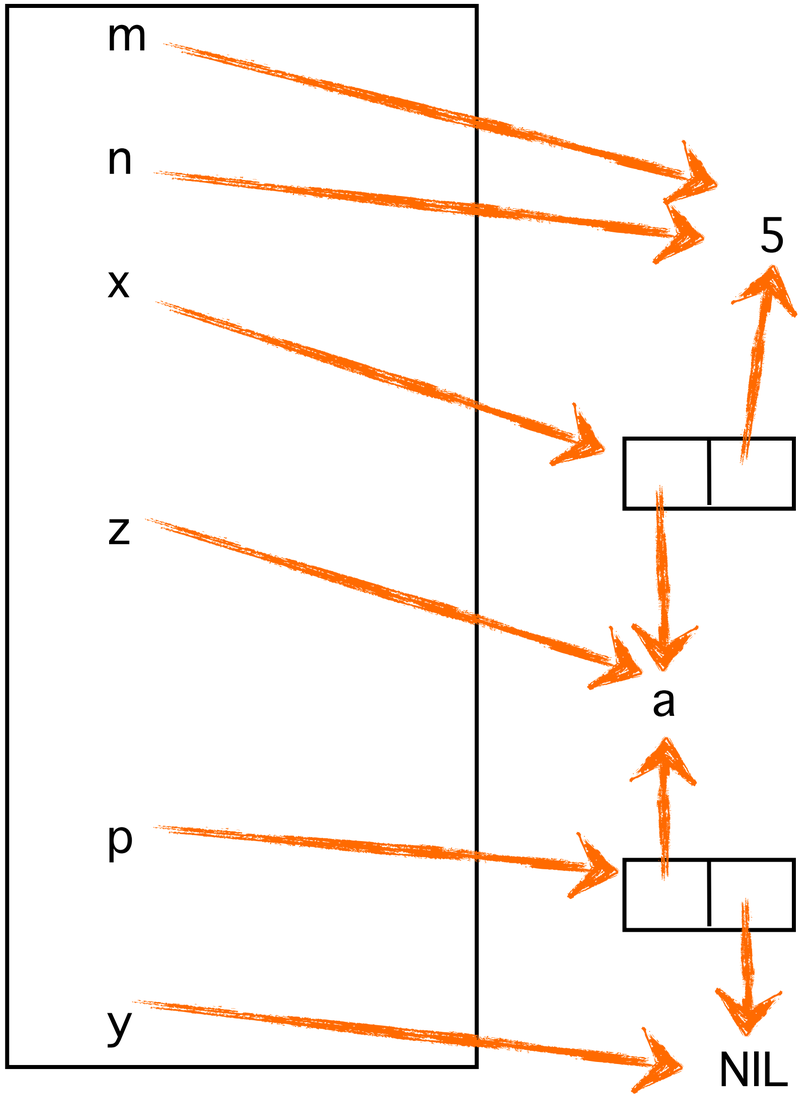
```
(define (insert-first! e lst)
   (set! lst (cons e lst)))
```

```
> (define test '(1 2 3))
> test
(1 2 3)
> (insert-first! 4 test)
> test
(1 2 3)
```

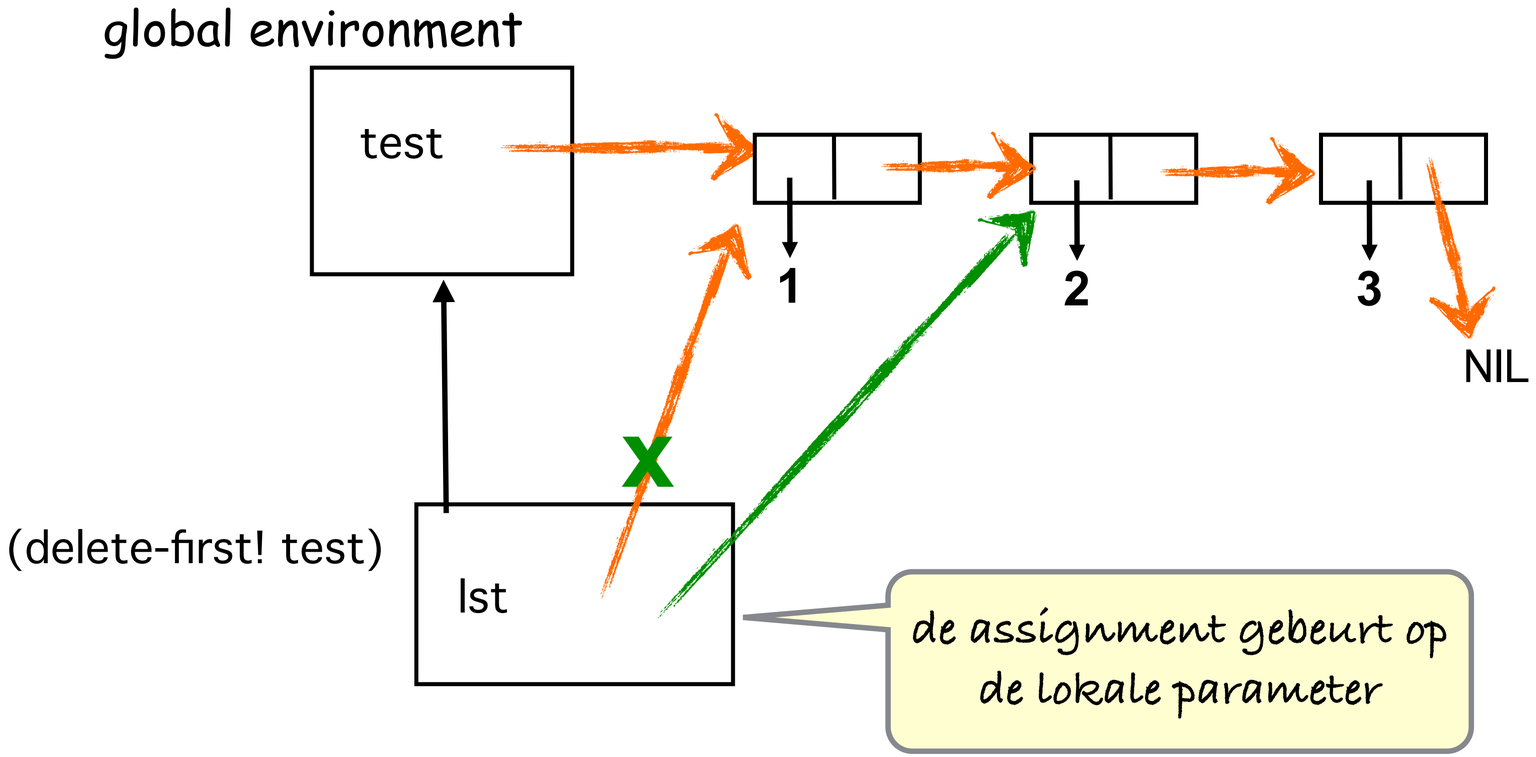
!!! call by value staat in de weg

“Bindingen” herbekeken





Call by value maakt het onmogelijk om via set! een actuele parameter aan te passen



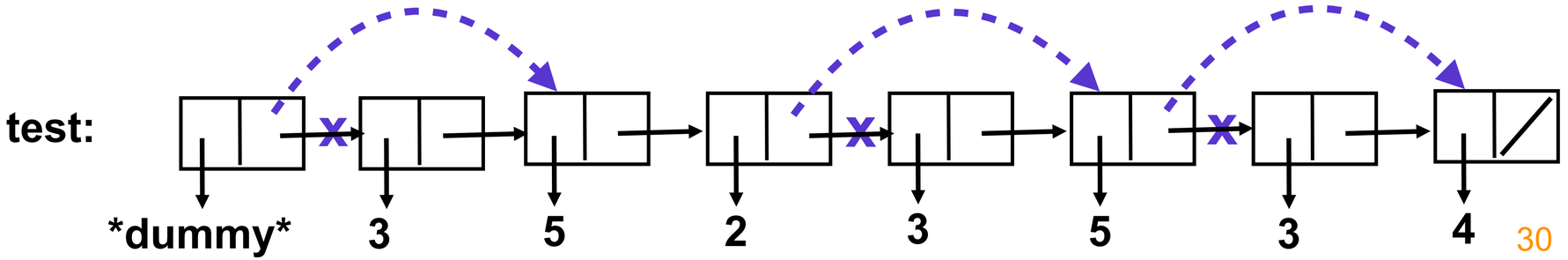
Headed-lists

standaard oplossing om problemen met eerste cel te vermijden bij destructieve lijstmanipulaties

```
(define (make-headed-list lst)
   (cons *dummy* lst))
```

```
> (define test (make-headed-list '(3 5 2 3 5 3 4)))
> test
(*dummy* 3 5 2 3 5 3 4)
> (remove! 3 test)
done
> test
(*dummy* 5 2 5 4)
```

remove! werkt nu wel omdat het te deleten getal nooit meer op de eerste plaats kan staan



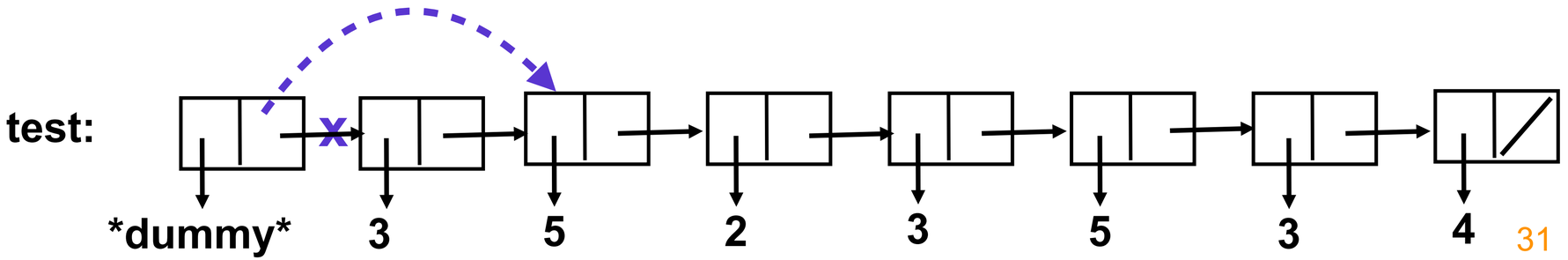
Headed-lists (2)

!!! implementaties moeten iha wel aangepast worden

```
(define (delete-first! lst)
   (set! lst (cdr lst)))
```

```
(define (delete-first! lst)
   (set-cdr! lst (cdr (cdr  lst))))
```

```
> (define test (make-headed-list
'(3 5 2 3 5 3 4)))
> test
(*dummy* 3 5 2 3 5 3 4)
> (delete-first! test)
done
> test
(*dummy* 5 2 3 5 3 4)
```



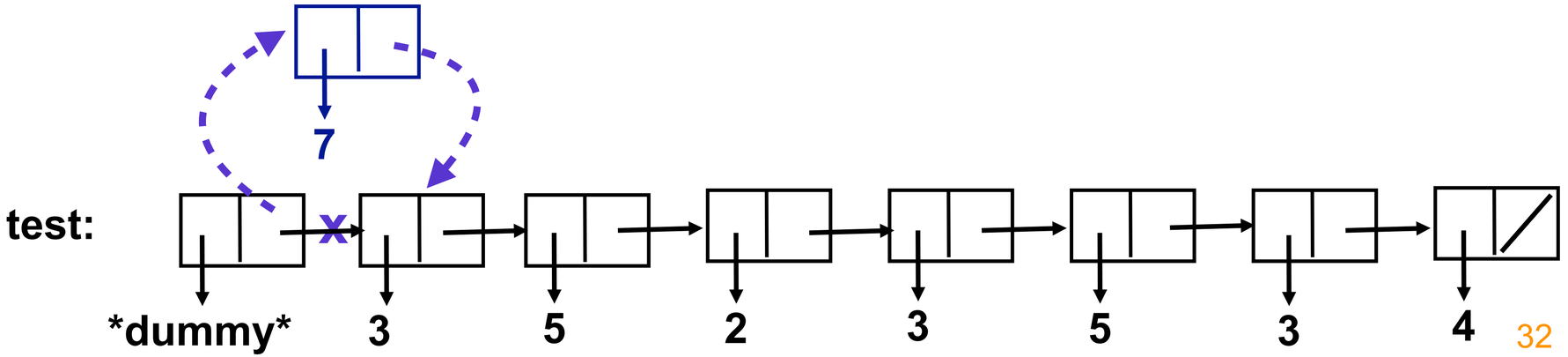
Headed-lists (3)

!!! implementaties moeten iha wel aangepast worden

```
(define (insert-first! e lst)
   (set! lst (cons e lst)))
```

```
(define (insert-first! e lst)
   (set-cdr! lst (cons e (cdr  lst))))
```

```
> (define test (make-headed-list '(3 5 2 3 5 3 4)))
> test
(*dummy* 3 5 2 3 5 3 4)
> (insert-first! 7 test)
done
> test
(*dummy* 7 3 5 2 3 5 3 4)
```



Les 11: dataabstractie stijlen