

# Les 5: Datastructureringsmechanismen

---

SESSIE 1

## Les 5: Datastructureringsmechanismen

---

Al de procedures die we tot nu toe schreven manipuleren eenvoudige numerieke data. In deze en de volgende lessen gaan we kijken naar meer complexe data. Waar de vorige lessen aandacht hadden voor combinatie- en abstractietechnieken voor procedure-objecten gaan we vanaf nu aandacht hebben voor combinatie- en abstractietechnieken voor data-objecten.

# Elementen van een programmeertaal

	Data	Procedure
Primitive	5    5.0 -3 3.14	+    - *    /
Combination		(* 3 5) (+ 1 2 3 4 5) (/ (* 3 5) (+ 1 2))
Abstraction	(define n 5) (define dubbel (lambda (x) (* 2 x)))	(define (gemiddelde x y) (/ (+ x y) 2))

# Datastructureringsmechanismen en data-abstractiemechanismen

Datastructureringsmechanismen zijn een soort lijm waarmee je kleinere data-objecten samenklit tot grotere complexe data-objecten.

Data-abstractie is dan een techniek om die delen van een programma die iets vertellen over hoe en waaruit complexe data objecten zijn samengesteld te isoleren van de andere delen van het programma die die complexe objecten manipuleren als een geheel.

Data-abstractie maakt het ontwerpen, onderhouden en aanpassen van een groot complex programma gemakkelijker.

# Overzicht

In het eerste deel komen eerst de basis datastructureringsmechanismen (het paar en de lijst) van Scheme aan bod.

In het tweede deel schrijven een aantal procedures die lijsten manipuleren om typische patronen voor lijstmanipulatie te schetsen.

In het derde deel maken we ook nog complexere combinaties van paren nml. geneste lijsten die als bomen kunnen geïnterpreteerd worden. We laten dan weer een aantal procedures zien die bomen manipuleren om typische patronen van boommanipulatie te schetsen.

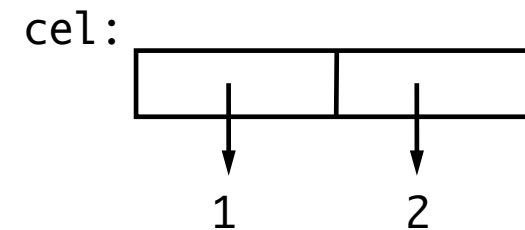
# Basis datastructureringsmechanisme – Paren

construeert  
een cons-cel

```
> (cons 1 2)
(1 . 2)
> (define cel (cons 1 2))
> cel
(1 . 2)
> (car cel)
1
> (cdr cel)
2
```

geeft de inhoud van  
het linker deel

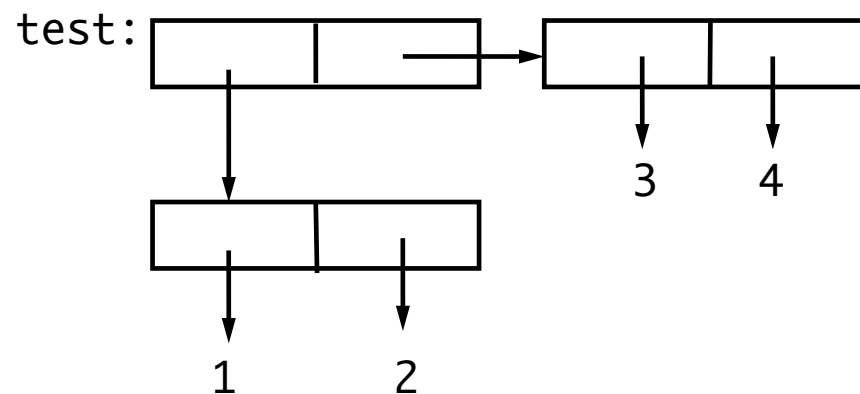
geeft de inhoud van  
het rechter deel



# Paren als universele bouwblokken (1)



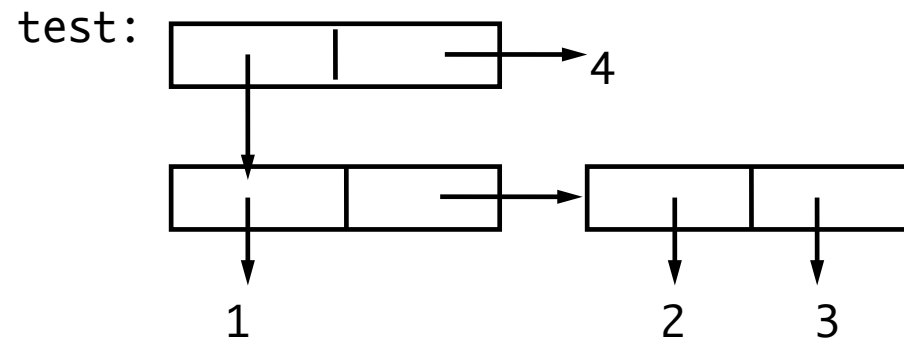
```
> (define test (cons (cons 1 2) (cons 3 4)))  
> test  
((1 . 2) 3 . 4)  
> (car (car test))  
1  
> (car (cdr test))  
3  
> (cdr (car test))  
2  
> (cdr (cdr test))  
4
```



## Paren als universele bouwblokken (2)



```
> (define test (cons (cons 1 (cons 2 3)) 4))  
> test  
((1 2 . 3) . 4)  
> (car (car test))  
1  
> (cdr test)  
4  
> (car (cdr (car test)))  
2  
> (cdr (cdr (car test)))  
3
```



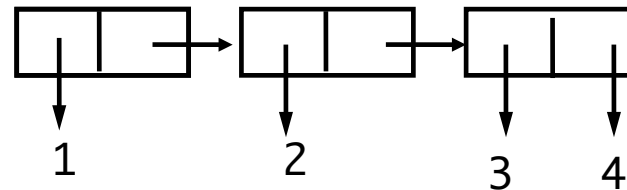


## Paren als universele bouwblokken (3)

```
> (define test (cons 1 (cons 2 (cons 3 4))))  
> test  
(1 2 3 . 4)  
> (car test)  
1  
> (car (cdr test))  
2  
> (car (cdr (cdr test)))  
3  
> (cdr (cdr (cdr test)))  
4
```

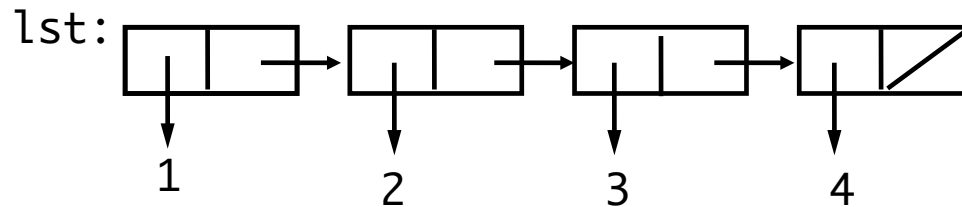


test:



# Datastructuringsmechanisme – Lijsten

```
> (define lst (cons 1 (cons 2 (cons 3 (cons 4 '())))))  
> lst  
(1 2 3 4)  
> (car lst)  
1  
> (car (cdr lst))  
2  
> (car (cdr (cdr lst)))  
3  
> (car (cdr (cdr (cdr lst))))  
4  
> (cdr lst)  
(2 3 4)  
> (cdr (cdr lst))  
(3 4)  
> (cdr (cdr (cdr lst)))  
(4)  
> (cdr (cdr (cdr (cdr lst))))  
()
```



# Lijsten – Constructors: cons, list, append



```
> (cons 1 (cons 2 (cons 3 (cons 4 '()))))  
(1 2 3 4)  
> (list 1 2 3 4)  
(1 2 3 4)
```

```
> (cons 1 2)  
(1 . 2)  
> (list 1 2)  
(1 2)  
> (cons 1 (cons 2 '()))  
(1 2)
```

let op! er is een verschil tussen een cons cel  
en een lijst met 2 elementen

```
> (append (list 1 2) (list 3 4))  
(1 2 3 4)  
> (define test (cons (cons 1 (cons (list 2 3) '())) (list (list 5) 6)))  
> test  
((1 (2 3)) (5) 6)
```

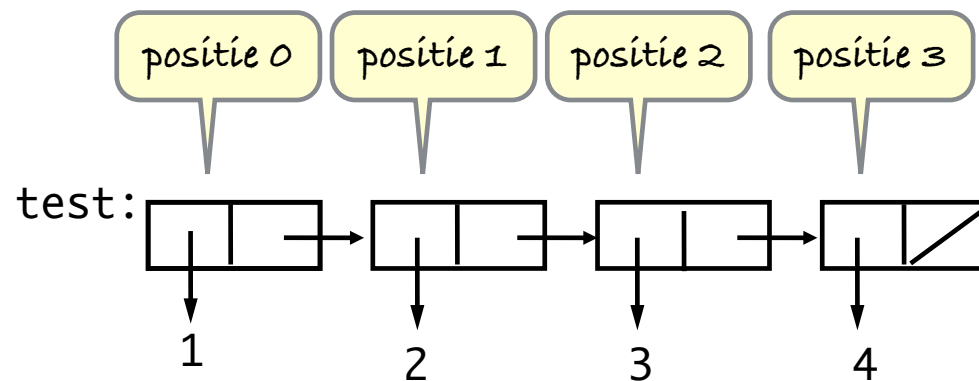
## Lijsten – Selectors; car, cdr en combinaties



```
> (define test
  (cons (cons 1 (cons (list 2 3) '())) (list (list 5) 6)))
> test
((1 (2 3)) (5) 6)
> (car test)
(1 (2 3))
> (cdr test)
((5) 6)
> (car (car test))
1
> (caar test)
1
> (cadr test)
(5)
> (cdar test)
((2 3))
> (cddr test)
()
```

# Lijsten - Meer operatoren

```
> (define test (list 1 2 3 4))  
> (length test)  
4  
> (length '())  
0  
> (reverse test)  
(4 3 2 1)  
> (null? test)  
#f  
> (null? '())  
#t  
> (pair? test)  
#t  
> (pair? '())  
#f  
> (list-ref test 2)  
3  
> (list-tail test 2)  
(3 4)
```



# Foutmeldingen

```
> (car 5)
⊗⊗ mcar: contract violation
expected: mpair?
given: 5
> (cdr 5)
⊗⊗ mcdr: contract violation
expected: mpair?
given: 5
> (car #t)
⊗⊗ mcar: contract violation
expected: mpair?
given: #t
> (cdr #f)
⊗⊗ mcdr: contract violation
expected: mpair?
given: #f
```

```
> (car '())
⊗⊗ mcar: contract violation
expected: mpair?
given: ()
> (cdr '())
⊗⊗ mcdr: contract violation
expected: mpair?
given: ()
> (list-ref test 7)
⊗⊗ mcdr: contract violation
expected: mpair?
given: ()
> (list-tail test 7)
⊗⊗ mcdr: contract violation
expected: mpair?
given: ()
```



# Lijstoperatoren zijn functies!



```
> (define test (list 1 2 3 4))
> (length test)
4
> test
(1 2 3 4)
> (reverse test)
(4 3 2 1)
> test
(1 2 3 4)
> (cons 0 test)
(0 1 2 3 4)
> test
(1 2 3 4)
> (cdr test)
(2 3 4)
> test
(1 2 3 4)
```

```
> (define lst1 (list 1 2 3))
> (define lst2 (list 4 5))
> (append lst1 lst2)
(1 2 3 4 5)
> lst1
(1 2 3)
> lst2
(4 5)

> (define a 3)
> (define b 5)
> (+ a b)
8
> a
3
> b
5
```

een functie produceert een  
resultaat maar tast haar  
argumenten niet aan

# Les 5: Datastructureringsmechanismen

---

SESSIE 2



# Lineair recursieve processen over lijsten (1)



```
(define (mylength lst)
  (if (null? lst)
      0
      (+ 1 (mylength (cdr lst)))))
```

```
> test
(1 2 3 4)
> (mylength test)
4
```

```
> (mylength test)
> (mylength (mcons 1 (mcons 2 (mcons 3 (mcons 4 '())))))
> (mylength (mcons 2 (mcons 3 (mcons 4 '()))))
> > (mylength (mcons 3 (mcons 4 '())))
> > (mylength (mcons 4 '()))
> > > (mylength '())
< < < 0
< < 1
< < 2
< 3
< 4
4
```

de lege lijst bereiken is  
de eindconditie

de lijst wordt in elke  
stap ingekort

opbouwen van het resultaat  
(constructieve recursie)

## Lineair recursieve processen over lijsten (2)



```
(define (sum-all lst)
  (if (null? lst)
      0
      (+ (car lst) (sum-all (cdr lst)))))
```

```
> test
(1 2 3 4)
> (sum-all test)
10
```

```
> (sum-all test)
> (sum-all (mcons 1 (mcons 2 (mcons 3 (mcons 4 '())))))
> (sum-all (mcons 2 (mcons 3 (mcons 4 '()))))
> > (sum-all (mcons 3 (mcons 4 '())))
> > (sum-all (mcons 4 '()))
> > > (sum-all '())
< < < 0
< < 4
< < 7
< 9
< 10
10
```

0 is neutraal element  
voor de optelling

opbouwen van het  
resultaat

## Lineair recursieve processen over lijsten (3)

lege lijst is  
neutraal  
element voor  
de cons  
operator

```
(define (square-all lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (square-all (cdr lst)))))
```

```
> test
(1 2 3 4)
> (square-all test)
(1 4 9 16)
```

```
> (square-all test)
> (square-all (mcons 1 (mcons 2 (mcons 3 (mcons 4 '())))))
> (square-all (mcons 2 (mcons 3 (mcons 4 '()))))
> > (square-all (mcons 3 (mcons 4 '())))
> > (square-all (mcons 4 '()))
> > > (square-all '())
< < < '()
< < (mcons 16 '())
< < (mcons 9 (mcons 16 '()))
< (mcons 4 (mcons 9 (mcons 16 '())))
< (mcons 1 (mcons 4 (mcons 9 (mcons 16 '()))))
(1 4 9 16)
```



## Lineair recursieve processen over lijsten (4)

```
(define (myreverse lst)
  (if (null? lst)
      '()
      (append (myreverse (cdr lst))
                (list (car lst)))))
```

truukje om achteraan  
toe te voegen

```
> test
(1 2 3 4)
> (myreverse test)
(4 3 2 1)
```

```
> (myreverse test)
> (myreverse (mcons 1 (mcons 2 (mcons 3 (mcons 4 '())))))
> (myreverse (mcons 2 (mcons 3 (mcons 4 '()))))
> > (myreverse (mcons 3 (mcons 4 '())))
> > (myreverse (mcons 4 '()))
> > > (myreverse '())
< < < '()
< < (mcons 4 '())
< < (mcons 4 (mcons 3 '()))
< (mcons 4 (mcons 3 (mcons 2 '())))
< (mcons 4 (mcons 3 (mcons 2 (mcons 1 '()))))
(4 3 2 1)
```

lege lijst is neutraal element  
voor de append operator



## Zoeken in lijsten (1)

```
(define (member1 el lst)
  (if (null? lst)
      #f
      (or (= el (car lst))
          (member1 el (cdr lst)))))
```

```
> test
(1 2 3 4)
> (member1 3 test)
#t
> (member1 7 test)
#f
```

```
> (member1 3 test)
>(member1 3 (mcons 1 (mcons 2 (mcons 3 (mcons 4 '())))))
>(member1 3 (mcons 2 (mcons 3 (mcons 4 '()))))
>(member1 3 (mcons 3 (mcons 4 '())))
<#t
#t
> (member1 7 test)
>(member1 7 (mcons 1 (mcons 2 (mcons 3 (mcons 4 '())))))
>(member1 7 (mcons 2 (mcons 3 (mcons 4 '()))))
>(member1 7 (mcons 3 (mcons 4 '())))
>(member1 7 (mcons 4 '()))
>(member1 7 '())
<#f
#f
```

eens de 3 is gevonden  
wordt de rest van de lijst  
niet meer bezocht

dat 7 niet in de lijst zit weet  
je pas als je heel de lijst hebt  
bekeken

## Zoeken in lijsten (2)

```
(define (member2 el lst)
  (cond
    ((null? lst) #f)
    ((= el (car lst)) #t)
    (else (member2 el (cdr lst)))))
```

```
> test
(1 2 3 4)
> (member2 3 test)
#t
> (member2 7 test)
#f
```

```
> (member2 3 test)
>(member2 3 (mcons 1 (mcons 2 (mcons 3 (mcons 4 '())))))
>(member2 3 (mcons 2 (mcons 3 (mcons 4 '()))))
>(member2 3 (mcons 3 (mcons 4 '())))
<#t
#t
> (member2 7 test)
>(member2 7 (mcons 1 (mcons 2 (mcons 3 (mcons 4 '())))))
>(member2 7 (mcons 2 (mcons 3 (mcons 4 '()))))
>(member2 7 (mcons 3 (mcons 4 '())))
>(member2 7 (mcons 4 '()))
>(member2 7 '())
<#f
#f
```

net zelfde traces als bij  
vorige versie

OR en AND zijn special forms  
Ze zijn 'lui'

(and <expression-1> . . . <expression-n>)

! zo gauw één deexpressie FALSE geeft worden  
de verdere deexpressies niet meer geëvalueerd

(or <expression-1> . . . <expression-n>)

! zo gauw één deexpressie TRUE geeft worden  
de verdere deexpressies niet meer geëvalueerd

## Mappen over lijsten

```
(define (square-all lst)
  (if (null? lst)
      '()
      (cons (square (car lst))
            (square-all (cdr lst)))))
```

```
(define (double-all lst)
  (if (null? lst)
      '()
      (cons (double (car lst))
            (double-all (cdr lst)))))
```

```
(define (cube-all lst factor)
  (if (null? lst)
      '()
      (cons (cube (car lst) factor)
            (cube-all (cdr lst) factor))))
```

$(e_1 \ e_2 \ \dots \ e_n)$



$(f(e_1) \ f(e_2) \ \dots \ f(e_n))$

```
(define (mymap f lst)
  (if (null? lst)
      '()
      (cons (f (car lst))
            (mymap f (cdr lst)))))
```

```
> test
(1 2 3 4)
> (mymap square test)
(1 4 9 16)
> (mymap double test)
(2 4 6 8)
> (mymap cube test)
(1 8 27 64)
```



# Mappen over lijsten (voorbeelden)

```
> (mymap square test)
(1 4 9 16)
> (mymap double test)
(2 4 6 8)
> (mymap cube test)
(1 8 27 64)
```

```
> (mymap (lambda (x) (* x x)) test)
(1 4 9 16)
> (mymap (lambda (x) (* x 2)) test)
(2 4 6 8)
> (mymap (lambda (x) (* x x x)) test)
(1 8 27 64)
```

```
(define (scale-list lst factor)
  (if (null? lst)
      '()
      (cons (* (car lst) factor)
            (scale-list (cdr lst) factor))))
```

```
> test
(1 2 3 4)
> (scale-list test 5)
(5 10 15 20)
```

```
(define (scale-list lst factor)
  (mymap (lambda (x) (* x factor)) lst))
```

```
> test
(1 2 3 4)
> (scale-list test 5)
(5 10 15 20)
```

## Accumuleren over lijsten

$(e_1 e_2 \dots e_n)$



$e_1 \text{ op } e_2 \text{ op } \dots \text{ op } e_n \text{ op } \boxed{ne}$

```
(define (sum-all lst)
  (if (null? lst)
      0
      (+ (car lst) (sum-all (cdr lst)))))
```

```
(define (mult-all lst)
  (if (null? lst)
      1
      (* (car lst) (mult-all (cdr lst)))))
```

```
(define (accumulate lst op ne)
  (if (null? lst)
      ne
      (op (car lst)
          (accumulate (cdr lst) op ne))))
```

```
> test
(1 2 3 4)
> (accumulate test + 0)
10
> (accumulate test * 1)
24
```

## Accumuleren over lijsten (voorbeelden)

```
(define (mylength lst)
  (if (null? lst)
      0
      (+ 1 (mylength (cdr lst)))))
```

```
(define (mylength lst)
  (accumulate
    (mymap (lambda (x) 1) lst) + 0))
```

```
(define (flatten lst)
  (if (null? lst)
      '()
      (append (car lst)
               (flatten (cdr lst)))))
```

```
(define (flatten lst)
  (accumulate lst append '()))
```

```
> test
(1 2 3 4)
> (mymap (lambda (x) 1) test)
(1 1 1 1)
> (accumulate
    (mymap (lambda (x) 1) test) + 0)
4
```

```
> (define duos
    (list (list 1 2) (list 3 4) (list 5 6)))
> duos
((1 2) (3 4) (5 6))
> (accumulate duos append '())
(1 2 3 4 5 6)
```


## Filteren van lijsten

```
(define (filter-odd lst)
  (cond ((null? lst) '())
        ((odd? (car lst))
         (cons (car lst) (filter-odd (cdr lst))))
        (else (filter-odd (cdr lst)))))
```

```
(define (filter-pos lst)
  (cond ((null? lst) '())
        ((> (car lst) 0)
         (cons (car lst) (filter-pos (cdr lst))))
        (else (filter-pos (cdr lst)))))
```

$(e_1 \ e_2 \ e_3 \ e_4 \ e_5 \ \dots \ e_n)$

test



$(e_1 \ e_2 \ \cancel{e_3} \ e_4 \ \cancel{e_5} \ \dots \ e_n)$

```
(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst))
         (cons (car lst) (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

```
> test
(1 -2 -3 4 5 -6 7 8 -9)
> (filter odd? test)
(1 3 5 7 9)
> (filter (lambda (x) (> x 0)) test)
(1 4 5 7 8)
```

# Les 5: Datastructureringsmechanismen

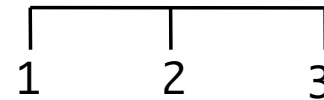
---

SESSIE 3

# Bomen als geneste lijsten

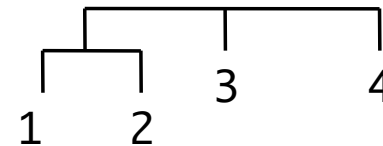
```
> (list 1 2 3)  
(1 2 3)
```

een lijst met 3  
getallen



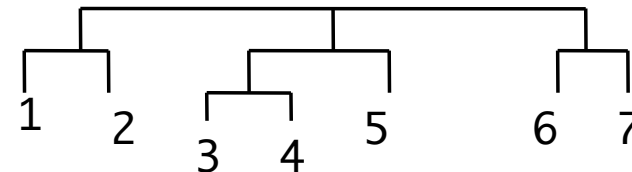
```
> (list (list 1 2) 3 4)  
((1 2) 3 4)
```

een lijst met 3  
elementen



```
> (list (list 1 2) (cons (list 3 4) (list 5)) (list 6 7))  
((1 2) ((3 4) 5) (6 7))
```

ook een lijst met 3  
elementen



```
> (define test (list (list 1 2) (cons (list 3 4) (list 5)) (list 6 7)))  
> (length test)  
3
```

# Bomen doorlopen: Tellen

```
(define (atom? test) (not (pair? test)))  
  
(define (count tree)  
  (cond  
    ((null? tree) 0)  
    ((atom? tree) 1)  
    (else (+ (count (car tree))  
              (count (cdr tree))))))
```

twee verschillende  
eindcondities: als je  
herhaaldelijk car en cdr  
gebruikt eindig je OF  
op een lege lijst, OF op  
een 'atomair' element

boomrecursie: tel in de car,  
tel in de cdr, en tel de  
resultaten samen

```
> test  
((1 2) ((3 4) 5) (6 7))  
> (length test)  
3  
> (count test)  
7
```

# Trace van count

```
> (count '((1 2) ((3 4) 5) (6 7)))  
>(count ((1 2) ((3 4) 5) (6 7)))  
> (count (1 2))  
> >(count 1)  
< <1  
> >(count (2))  
> > (count 2)  
< < 1  
> > (count ())  
< < 0  
< <1  
< 2  
> (count (((3 4) 5) (6 7)))  
> >(count ((3 4) 5))  
> > (count (3 4))  
> > >(count 3)  
< < <1  
> > >(count (4))  
> > > (count 4)  
< < < 1  
> > > (count ())  
< < < 0  
< < <1  
< < 2
```

```
> > (count (5))  
> > >(count 5)  
< < <1  
> > >(count ())  
< < <0  
< < 1  
< <3  
> >(count ((6 7)))  
> > (count (6 7))  
> > >(count 6)  
< < <1  
> > >(count (7))  
> > > (count 7)  
< < < 1  
> > > (count ())  
< < < 0  
< < <1  
< < 2  
> > (count ())  
< < 0  
< <2  
< 5  
<7
```

7



# Bomen doorlopen: Spiegelen

```
(define (mirror tree)
  (cond
    ((null? tree) tree)
    ((atom? tree) tree)
    (else (append (mirror (cdr tree))
                  (list (mirror (car tree)))))))
```

```
> test
((1 2) ((3 4) 5) (6 7))
> (reverse test)
((6 7) ((3 4) 5) (1 2))
> (mirror test)
((7 6) (5 (4 3)) (2 1))
```

truukje om element  
achteraan aan een lijst toe  
te voegen

# Trace van mirror

```
> (mirror '((1 2) ((3 4) 5) (6 7)))
>(mirror ((1 2) ((3 4) 5) (6 7)))
> (mirror (((3 4) 5) (6 7)))
> >(mirror ((6 7)))
> > (mirror ())
< < ()
> > (mirror (6 7))
> > >(mirror (7))
> > > (mirror ())
< < < ()
> > > (mirror 7)
< < < 7
< < < (7)
> > >(mirror 6)
< < < 6
< < (7 6)
< < ((7 6))
> >(mirror ((3 4) 5))
> > (mirror (5))
> > >(mirror ())
< < < ()
> > >(mirror 5)
< < < 5
< < (5)
```

```
> > (mirror (3 4))
> > >(mirror (4))
> > > (mirror ())
< < < ()
> > > (mirror 4)
< < < 4
< < < (4)
> > >(mirror 3)
< < < 3
< < (4 3)
< < (5 (4 3))
> ((7 6) (5 (4 3)))
> (mirror (1 2))
> >(mirror (2))
> > (mirror ())
< < ()
> > (mirror 2)
< < 2
< < (2)
> >(mirror 1)
< < 1
< (2 1)
< ((7 6) (5 (4 3)) (2 1))
((7 6) (5 (4 3)) (2 1))
```

## Bomen doorlopen: Zoeken

```
(define (deep-member el tree)
  (cond
    ((null? tree) #f)
    ((atom? tree) (eq? tree el))
    (else
     (or (deep-member el (car tree))
         (deep-member el (cdr tree))))))
```

onthou dat de OR 'lui' is

```
> test
((1 2) ((3 4) 5) (6 7))
> (member 4 test)
#f
> (deep-member 4 test)
#t
```

# Trace deep-member

```
> (deep-member 4 '((1 2) ((3 4) 5)
  (6 7)))
```

```
>(deep-member 4 ((1 2) ((3 4) 5)
  (6 7)))
```

```
> (deep-member 4 (1 2))
```

```
> >(deep-member 4 1)
```

```
< <#f
```

```
> (deep-member 4 (2))
```

```
> >(deep-member 4 2)
```

```
< <#f
```

```
> (deep-member 4 ())
```

```
< #f
```

```
>(deep-member 4 (((3 4) 5) (6 7)))
```

```
> (deep-member 4 ((3 4) 5))
```

```
> >(deep-member 4 (3 4))
```

```
> > (deep-member 4 3)
```

```
< < #f
```

```
> >(deep-member 4 (4))
```

```
> > (deep-member 4 4)
```

```
< < #t
```

```
< <#t
```

```
< #t
```

```
<#t
```

```
#t
```

eens de 4 is gevonden  
wordt de rest van de lijst  
niet meer bezocht

# Les 6: Data-abstractie

