Les 11: Data-abstractie stijlen

Sessie 1

Les 11: Data-abstractie stijlen

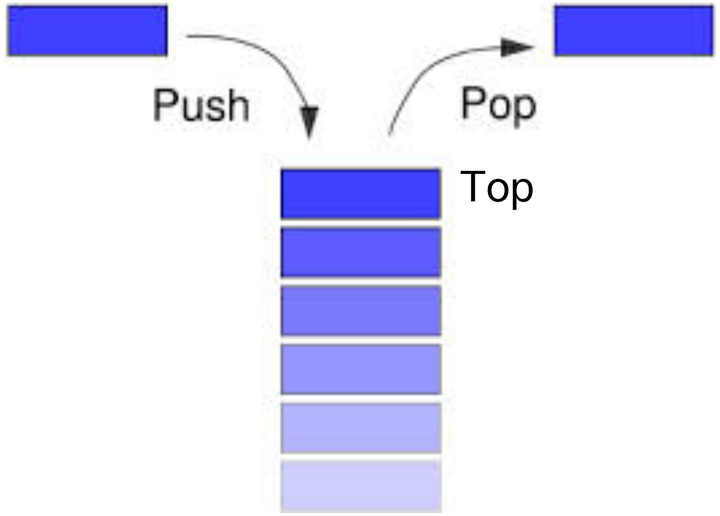
In deze les komen we terug op het onderwerp data-abstractie. We kennen ondertussen verschillende technieken die allen kunnen gebruikt worden bij het implementeren van een abstract data-type. We introduceren enkele eenvoudige abstracte data-types en tonen drie verschillende implementatiestijlen: functioneel, destructief, en object-geörienteerd.

Overzicht

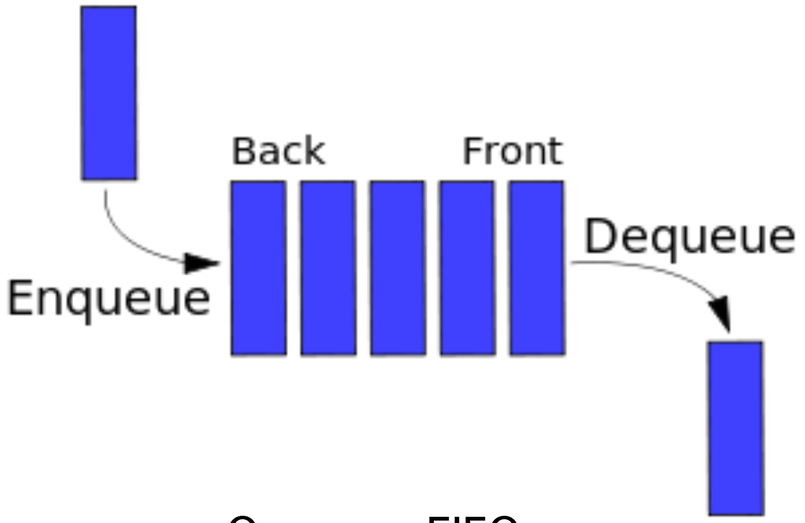
Als eerste voorbeelden worden de standaard ADT's **stack** en **queue** geïntroduceerd in verschillende stijlen.

Dan wordt het ADT **table** geïntroduceerd en als toepassing van het gebruik van procedure objecten tonen we de 'memoriseertechniek' voor het (potentieel) efficiënter maken van een functie.

Stack en Queue



Stack = LIFO

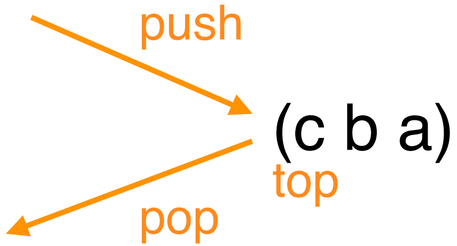


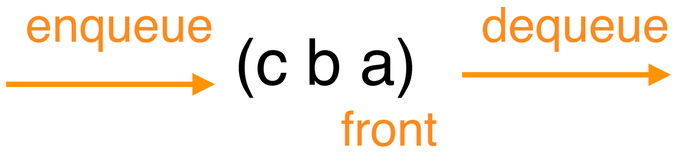
Queue = FIFO

ADT stack en queue - functionele versies

```
(define (create-stack)
  '())

(define (stack-empty? stack)
  (null? stack))

(define (push el stack )
  (cons el stack))

(define (pop stack)
  (cdr stack))

(define (top stack)
  (car stack))
```

```
(define (create-queue)
  '())

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

(define (enqueue el queue)
  (cons el queue))

(define (dequeue queue)
  (butlast queue))

(define (front queue)
  (car (last queue)))
```





ADT stack en queue - functionele versie: gebruik

```
> (define test (create-stack))
> test
()
> (push 'a test)
(a)
> test
()

> (set! test (push 'a test))
> test
(a)
> (set! test (push 'b test))
> (set! test (push 'c test))
> test
(c b a)
```

een 'nieuwe' stack met een extra element

de 'oude' stack is niet veranderd

de 'klant' moet zelf via assignment de nieuwe waarde vasthouden

ADT Stack - Mutator versie foute poging

```
(define (create-stack)
  '())

(define (stack-empty? stack)
  (null? stack))

(define (push! el stack)
  (set! stack (cons el stack)))

(define (pop! stack)
  (set! stack (cdr stack)))

(define (top stack)
  (car stack))
```

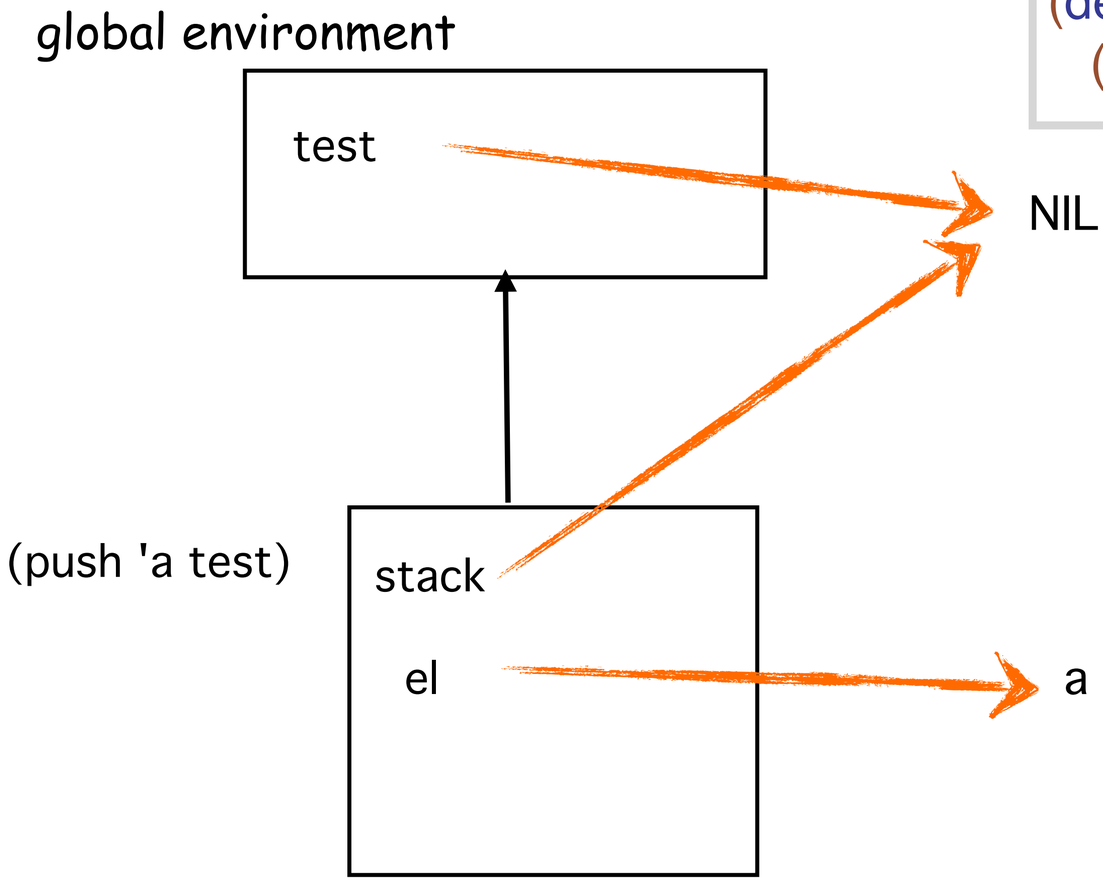
probeer met assignment in de body van push en pop

```
> (define test (create-stack))
> test
()
> (push! 'a test)
> test
()
```

Scheme gebruikt call-by-value als parameter passeer mechanisme; de 'oude' stack is niet aangepast

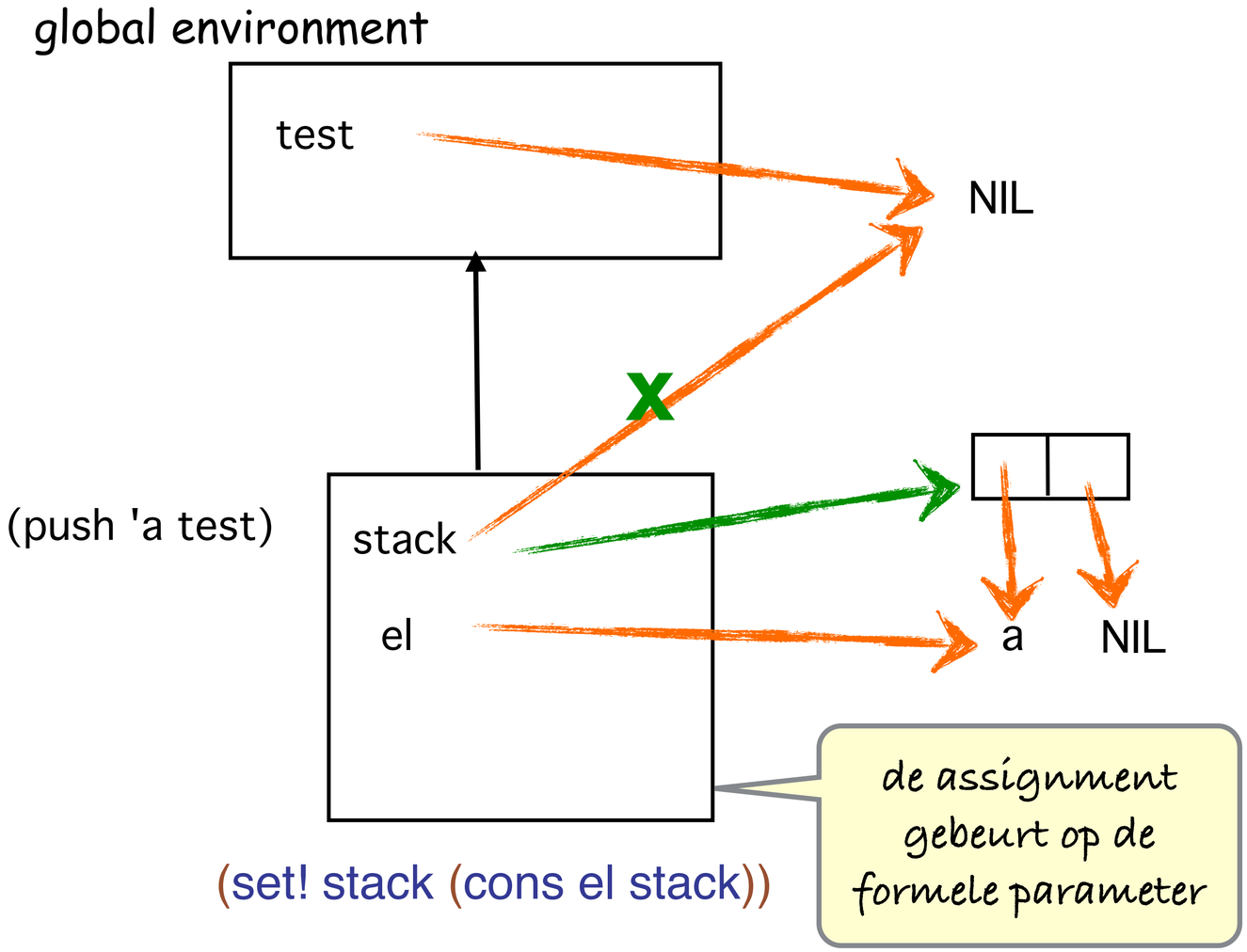
ADT Stack - Mutator versie
foute poging, omgevingsmodel

```
(define (push! el stack)
  (set! stack (cons el (car stack))))
```



(set! stack (cons el stack))

ADT Stack - Mutator versie
foute poging, omgevingsmodel



ADT Stack - Mutator versie correcte poging

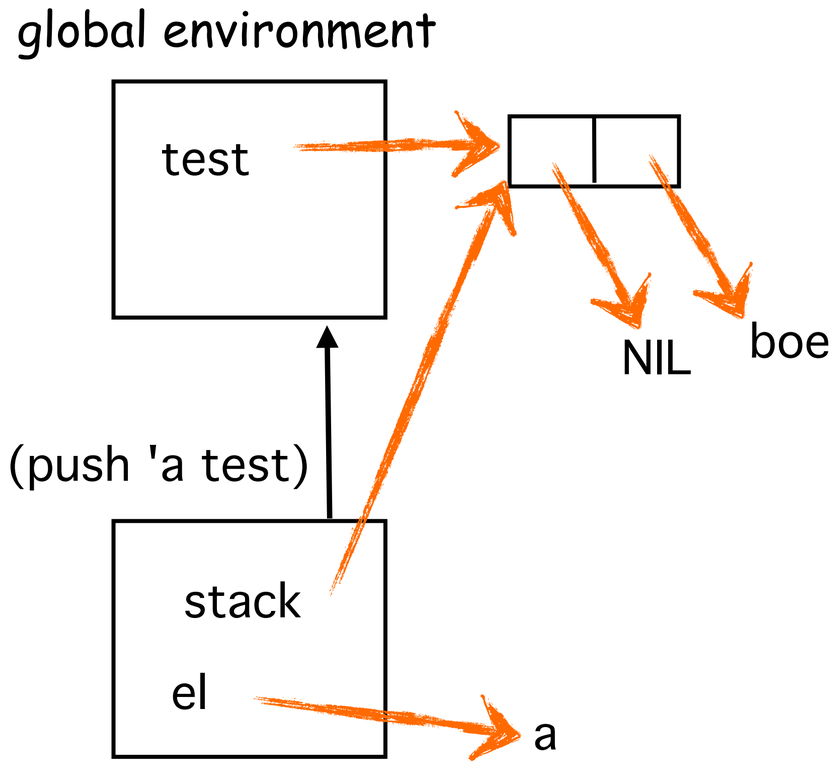
```
(define (create-stack)
  (cons '() 'boe))

(define (stack-empty? stack)
  (null? (car stack)))

(define (push! el stack)
  (set-car! stack (cons el (car stack))))

(define (pop! stack)
  (set-car! stack (cdr (car stack))))

(define (top stack)
  (car (car stack)))
```

maak een cons cel met de lege lijst in de car en om het even wat in de cdr

gebruik set-car! in de cons cel die de stack voorstelt om de lijst met stackelementen aan te passen

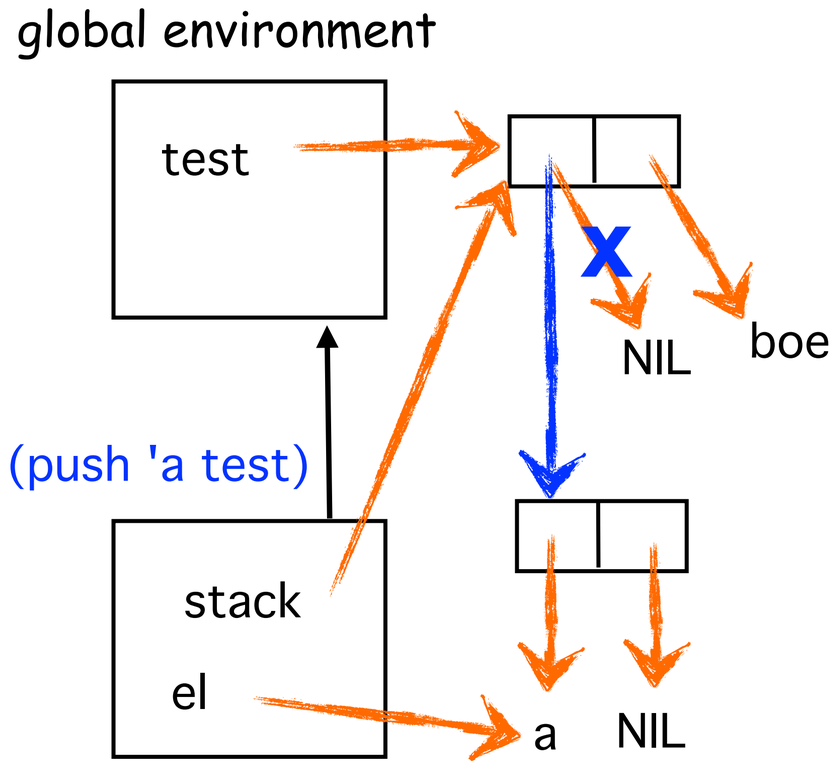
```
> (define test (create-stack))
> test
(() . boe)
> (push! 'a test)
> (push! 'b test)
> (push! 'c test)
> test
((c b a) . boe)
> (pop! test)
> test
((b a) . boe)
```

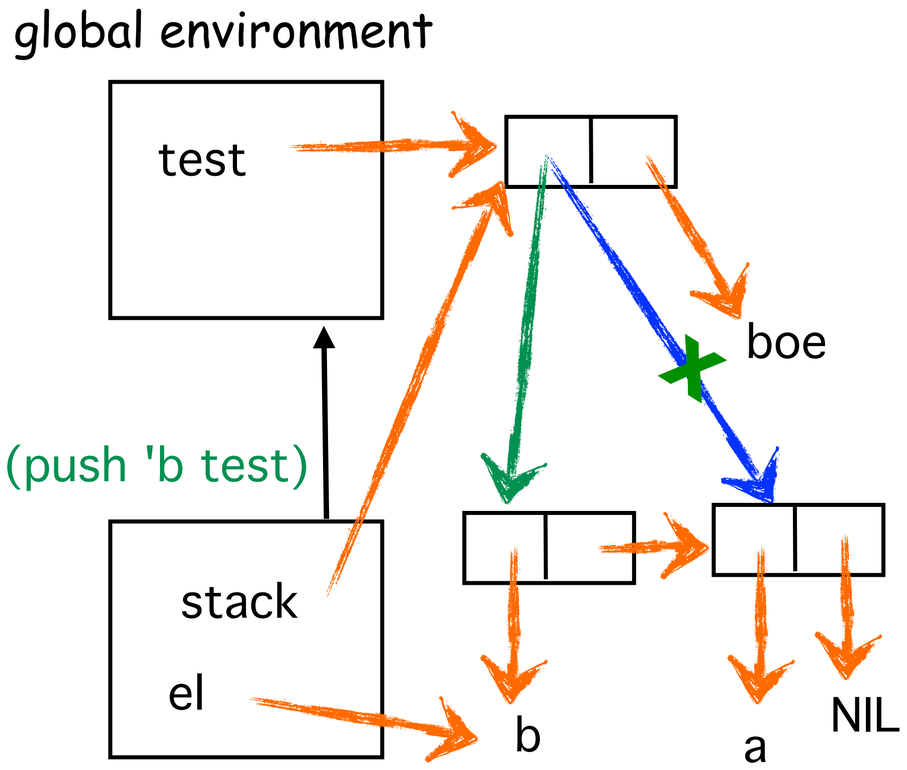
ADT Stack - Mutator versie
correcte poging, omgevingsmodel




```
(define (push! el stack)
  (set-car! stack (cons el (car stack))))
```

ADT Stack - Mutator versie
correcte poging, omgevingsmodel





ADT Stack - Mutator versie met extra abstractie

```
(define (make-pointer x)
  (cons x 'boe))

(define (create-stack)
  (make-pointer '()))

(define (stack-content stack)
  (car stack))

(define (set-stack-content! stack content)
  (set-car! stack content))

(define (stack-empty? stack)
  (null? (stack-content stack)))

(define (push! el stack)
  (set-stack-content! stack
     (cons el (stack-content stack))))

(define (pop! stack)
  (set-stack-content! stack
     (cdr (stack-content stack))))

(define (top stack)
  (car (stack-content stack)))
```

```
> (define test (create-stack))
> test
(() . boe)
> (push! 'a test)
> (push! 'b test)
> (push! 'c test)
> test
((c b a) . boe)
> (pop! test)
> test
((b a) . boe)
```

ADT Stack - OO versie

```
(define (make-stack)
  (let ((stack-content '()))
    (define (empty?)
      (null? stack-content))
    (define (push el)
      (set! stack-content (cons el stack-content)))
    (define (pop)
      (set! stack-content (cdr stack-content)))
    (define (top) (car stack-content))
    (define (dispatch m)
      (cond
        ((eq? m 'push) push)
        ((eq? m 'pop) pop)
        ((eq? m 'top) top)
        ((eq? m 'empty?) empty?)
        (else (error "unknown request
                -- MAKE-STACK" m))))
    dispatch))
```

assignment rechtstreeks op variable in scope

```
> (define test (make-stack))
> test
#<procedure>
> ((test 'push) ‘a)
> ((test 'push) 'b)
> test
#<procedure>
> ((test 'top))
b
> ((test 'pop))
> ((test 'top))
a
> ((test 'empty?))
#f
> ((test 'pop))
> ((test 'empty?))
#t
```

ADT queue - Mutator versie

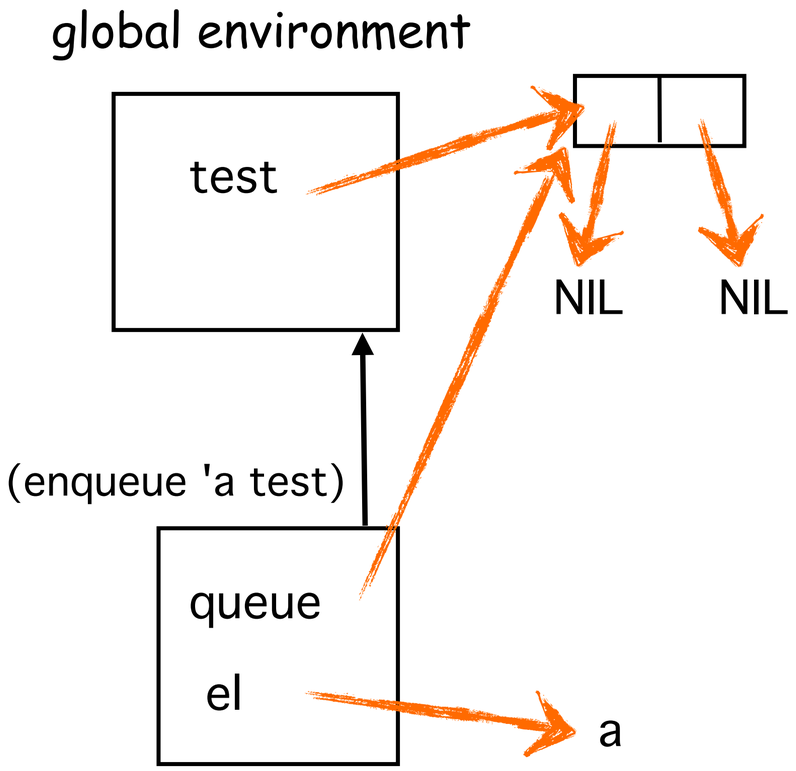
```
(define (create-queue) (cons '() '()))
(define (front-queue queue) (car queue))
(define (rear-queue queue) (cdr queue))
(define (set-front-queue! queue cell)
   (set-car! queue cell))
(define (set-rear-queue! queue cell)
    (set-cdr! queue cell))
(define (queue-empty? queue)
    (null? (front-queue queue)))
(define (enqueue el queue)
  (let ((cell (cons el '())))
    (cond ((queue-empty? queue)
           (set-front-queue! queue cell)
           (set-rear-queue! queue cell))
          (else
           (set-cdr! (rear-queue queue) cell)
           (set-rear-queue! queue cell)))))
(define (serve queue)
  (let ((front (front-queue queue)))
    (set-front-queue! queue
       (cdr (front-queue queue))
    (car front)))
```

zowel car als cdr worden nuttig gebruikt: car wijst naar het eerste element, cdr naar het laatste

```
> (define test (create-queue))
> (enqueue 'a test)
> (enqueue 'b test)
> test
((a b) b)
> (queue-empty? test)
#f
> (serve test)
a
```

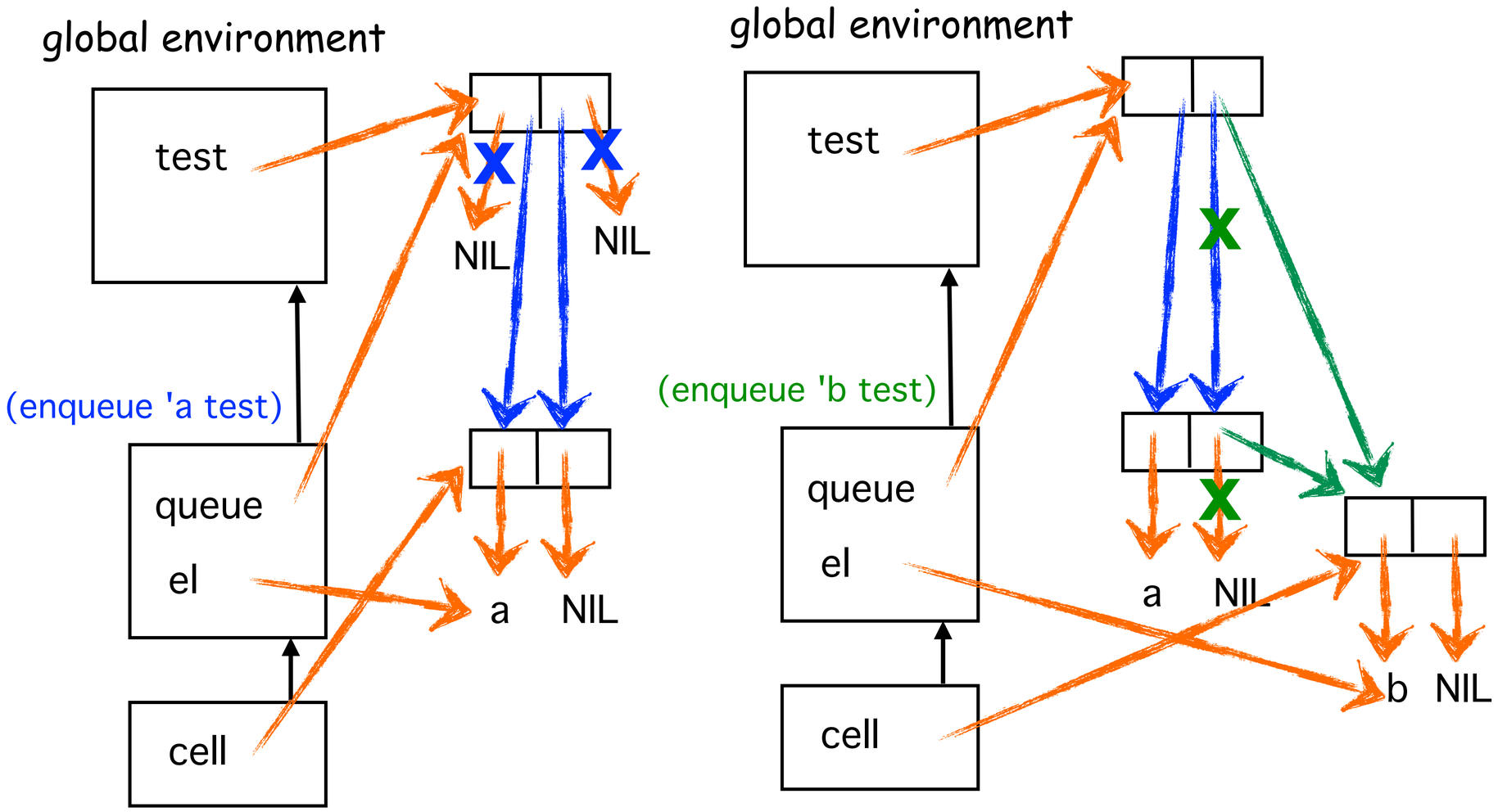
in deze implementatie gebeuren dequeue en het teruggeven van de front in 1 serve beweging

ADT queue - Mutator versie

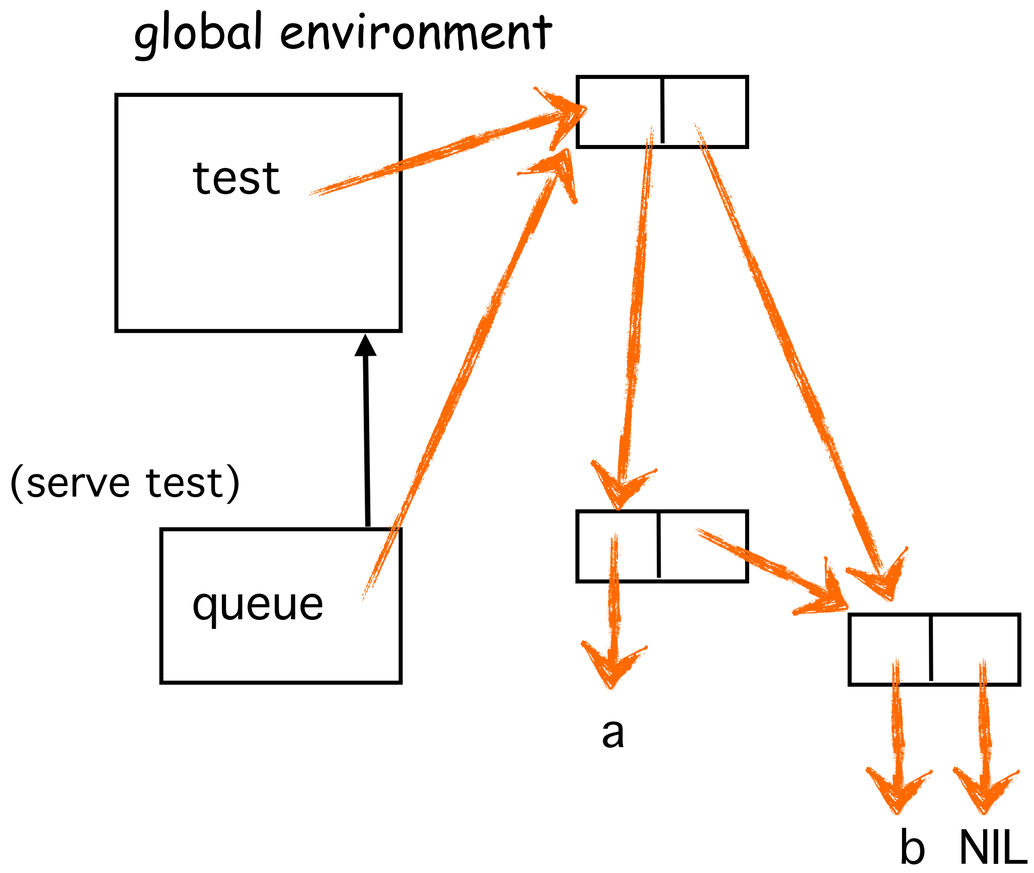



```
(define (enqueue el queue)
  (let ((cell (cons el '())))
    (cond ((queue-empty? queue)
           (set-front-queue! queue cell)
           (set-rear-queue! queue cell))
          (else
           (set-cdr! (rear-queue queue) cell)
           (set-rear-queue! queue cell)))))
```

ADT queue - Mutator versie

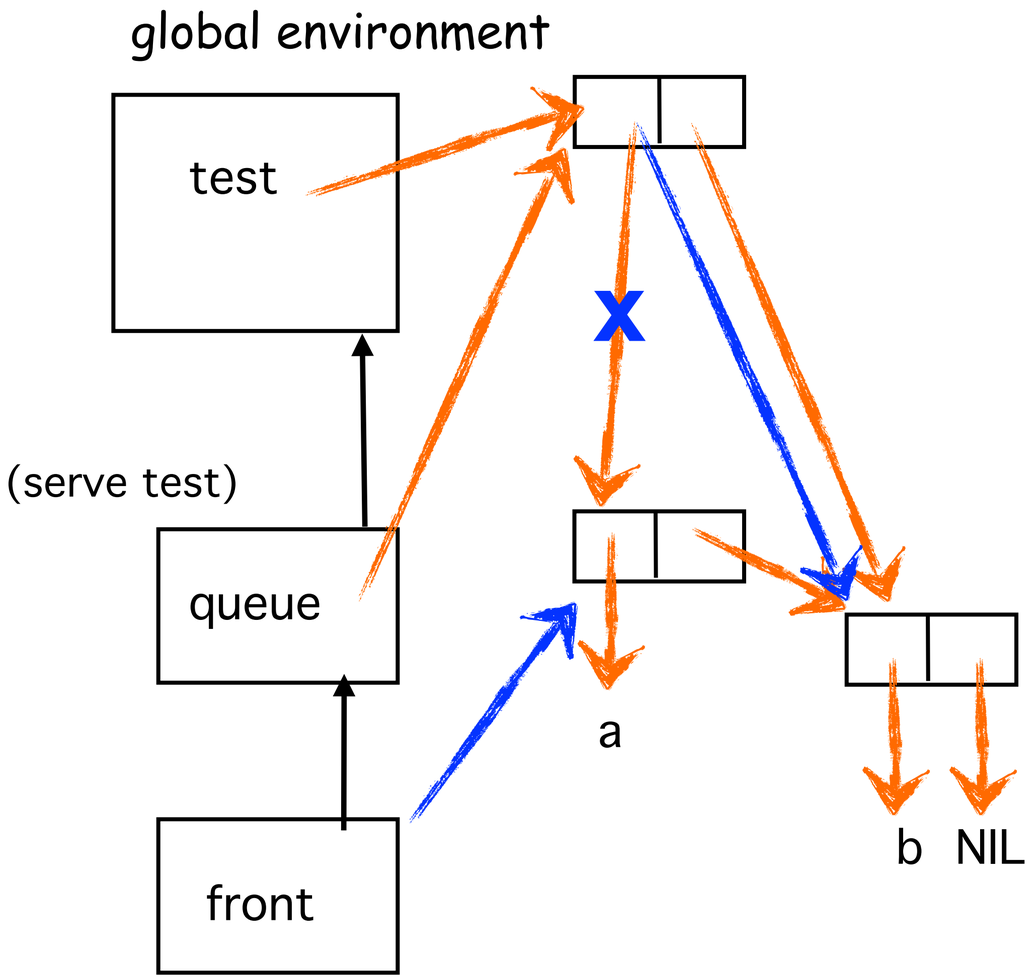


ADT queue - Mutator versie




```
(define (serve queue)
  (let ((front (front-queue queue)))
    (set-front-queue! queue
        (cdr (front-queue queue))
    (car front)))
```

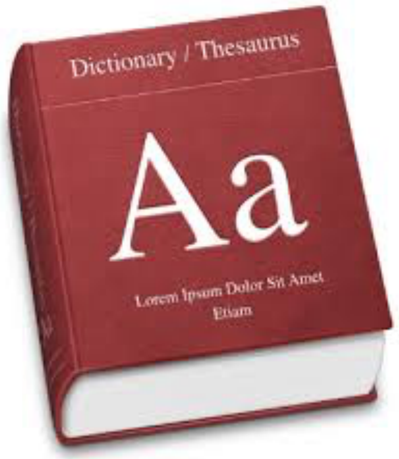
ADT queue - Mutator versie

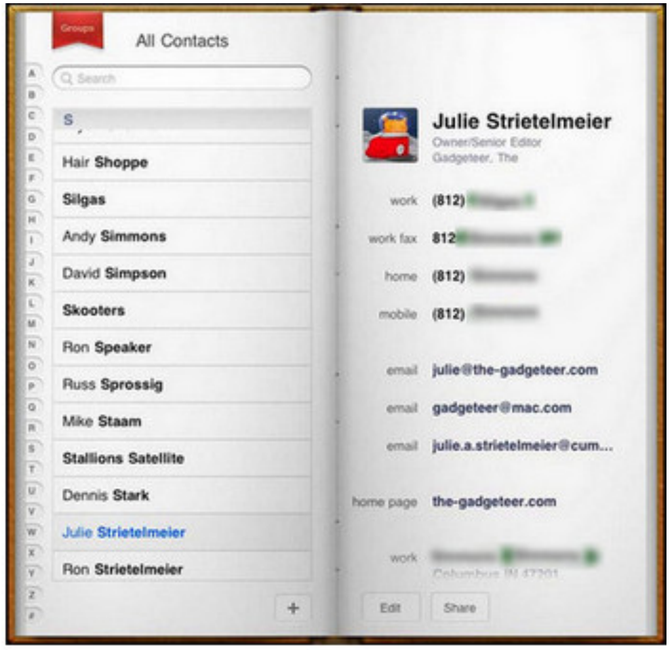


Les 11: Data-abstractie stijlen

Sessie 2

Dictionary - (Key/Value) Table - Map - Association





ADT Table - Mutator versie

```
(define (create-table) (cons '*table* '()))

(define (make-node key value)
  (cons key value))

(define (key node) (car node))

(define (value node) (cdr node))

(define (insert! key value table)
  (let ((node (make-node key value)))
    (set-cdr! table (cons node (cdr table)))))

(define (lookup key table)
  (let ((node (assq key (cdr table))))
    (if node
        (value node)
        #f)))
```

```
> (define test (create-table))
> test
(*table*)
> (insert! 'jan 222 test)
> (insert! 'an 333 test)
> test
(*table*(an . 333) (jan . 222))
> (lookup 'jan test)
222
> (lookup 'vivi test)
#f
> (lookup 'an test)
333
```

ADT Table - OO versie

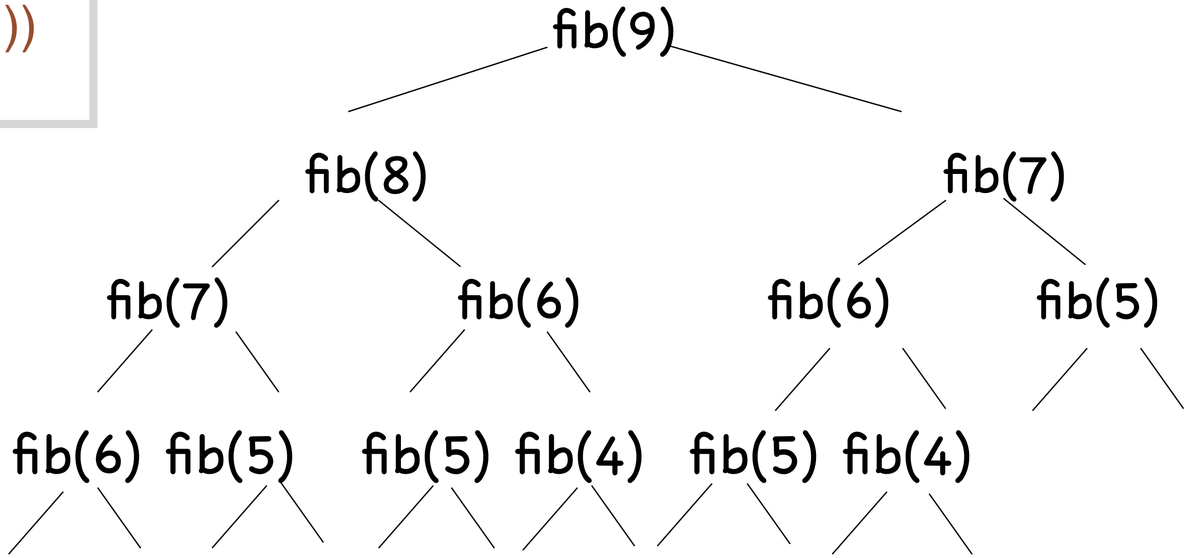
```
(define (make-table)
  (let ((table-content '()))
    (define (insert key value)
      (set! table-content
          (cons (make-node key value)
                table-content)))
    (define (lookup key)
      (let ((node (assq key table-content)))
        (if node
            (value node)
            #f)))
    (define (dispatch m)
      (cond ((eq? m 'insert) insert)
            ((eq? m 'lookup) lookup)))
     dispatch))
```

```
> (define test (make-table))
> test
#<procedure>
> ((test 'insert) 'jan 222)
> ((test 'insert) 'an 333)
> test
#<procedure>
> ((test 'lookup) 'an)
333
> ((test 'lookup) ‘vivi)
 #f
> ((test 'lookup) 'jan)
222
```

Fibonnaci herbekeken - Is niet echt efficiënt

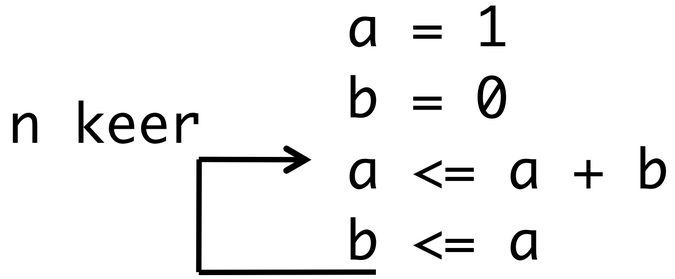
```
(define (fib n)
  (cond
    ((= n 0) 0)
    ((= n 1) 1)
    (else (+ (fib (- n 1))
             (fib (- n 2))))))
```

de fib van elk getal kleiner dan n wordt meer dan 1 keer berekend



fib(7) wordt 2 keer berekend
fib(6) wordt 3 keer berekend
fib(5) wordt 4 keer berekend ...

Fibonnaci herbekeken - Een iteratief process bestaat




```
a = fib(n)
b = fib(n -1)
```

```
(define (fib n)
  (define (fib-iter p q count)
    (if (= count 0)
        q
        (fib-iter (+ p q) p (- count 1))))
  (fib-iter 1 0 n))
```

is iha NIET zo dat er voor een boomrecursief process een iteratieve versie bestaat

De Memoize techniek

```
(define (memoize f)
  (let ((memo (create-table)))
    (lambda (x)
      (let ((old-result (lookup x memo)))
        (if old-result
            old-result
            (let ((result (f x)))
              (insert! x result memo)
              result))))))
```

sla resultaten voor reeds uitgerekende argumenten op in een tabel en zoek deze op i.p.v. de functie opnieuw uit te rekenen

```
> (define (sqr x) (* x x))
> sqr
#<procedure:sqr>
> (sqr 5)
25
> (define test (memoize sqr))
> test
#<procedure>
> (test 5)
25
> (test 7)
49
> (test 5)
old-result 25
```

trace om te bevestigen dat het een oud resultaat is

```
(if old-result
    (begin
      (display "old-result ")
      old-result)
```

Fibonnaci herbekeken met de Memoize techniek

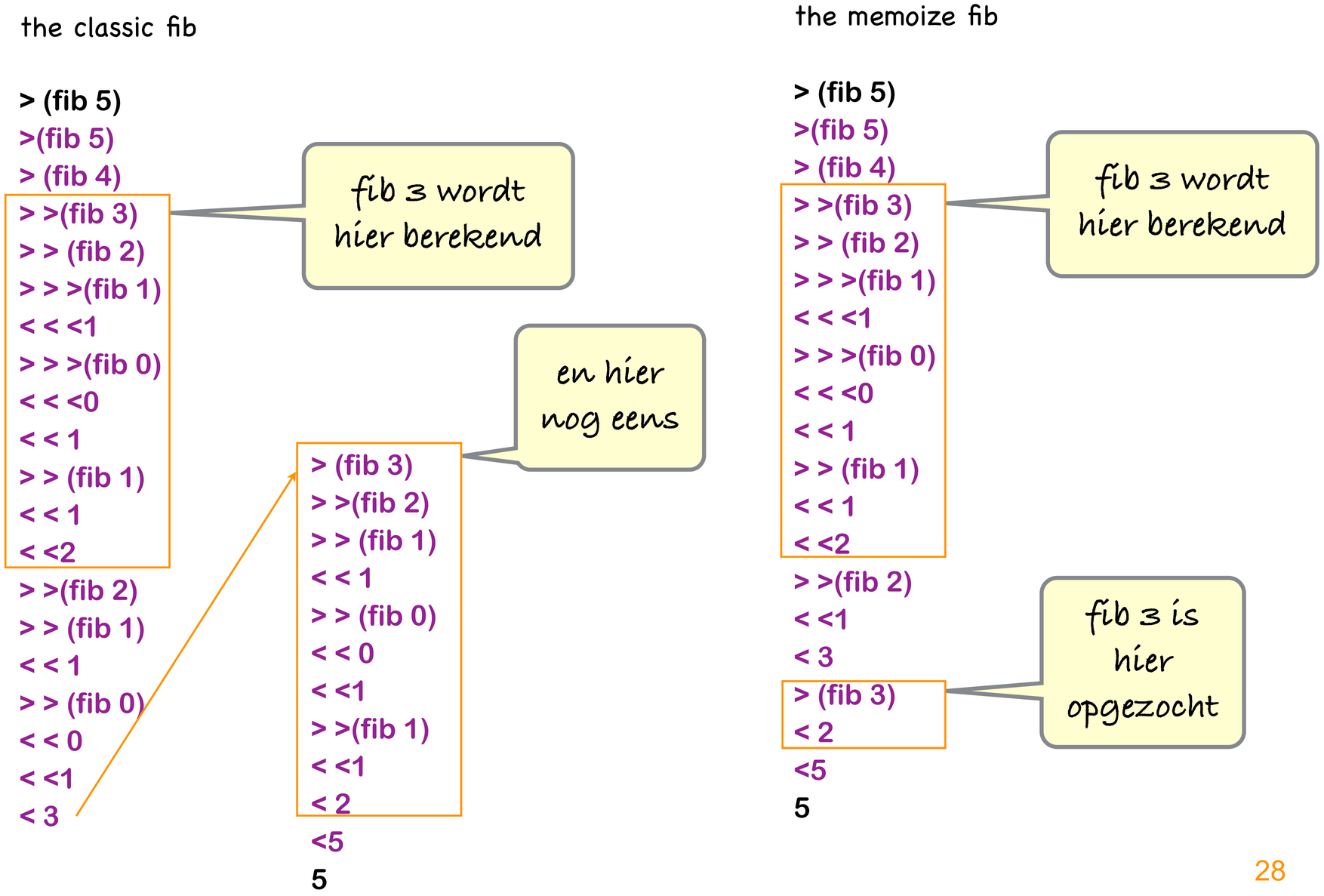
```
> fib
#<procedure:fib>
> (fib 5)
5
> (fib 9)
34

> (define fib (memoize fib))
> fib
#<procedure>
> (fib 5)
5
> (fib 9)
34
> (fib 7)
13
> (fib 4)
3
```

is algemene techniek die kan gebruikt worden bij overlappende subproblemen

om fib 9 te berekenen is fib 5 meerdere keren nodig, zal maar 1 keer berekend worden en daarna hergebruikt

fib 7 was al berekend tijdens de berekening van fib 9 en wordt nu rechtstreeks terugggeven

Fibonacci getallen: een trace



Les 12: Stromen