Les 12: Stromen

Sessie 1

Les 12: Stromen

In deze les worden 'stromen' geintroduceerd. We laten zien hoe het concept stroom toelaat om programma's elegant te structureren.

Om de volle kracht van stromen te kunnen exploiteren introduceren we ook het concept van uitgestelde evaluatie wat ons toelaat oneindig lange stromen te manipuleren.

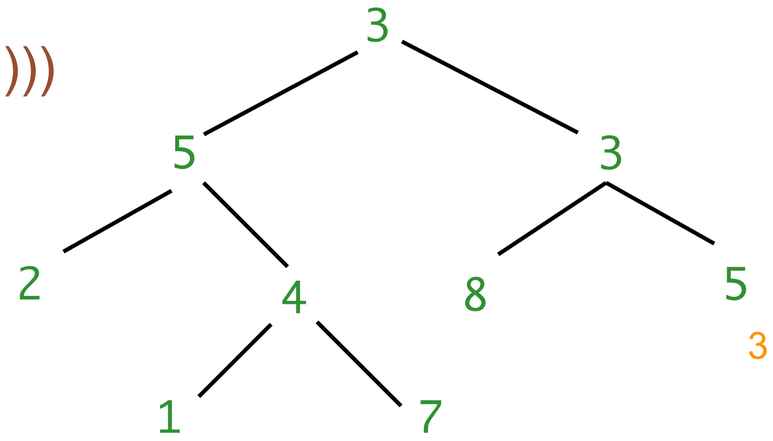
Stromen: introductie (1)

sum the squares of all odd leafs in a binary tree

```
(define (sum-sqr-odd tree)
  (if (leaf? tree)
      (if (odd? tree)
          (square tree)
          0)
      (+ (sum-sqr-odd (left tree))
          (sum-sqr-odd (right tree)))))
```

boomrecursie

```
> (define test '((2 5 (1 4 7)) 3 (8 3 5)))
> test
((2 5 (1 4 7)) 3 (8 3 5))
> (sum-sqr-odd test)
75
```



Stromen: introductie (2)

list all odd fibs of the numbers smaller than n

```
(define (odd-fibs n)
  (define (next k)
    (if (> k n)
        '()
        (let ((r (fib k)))
          (if (odd? r)
              (cons r (next (1+ k)))
              (next (1+ k))))))
  (next 1))
```

lineaire recursie

```
> (odd-fibs 5)
(1 1 3 5)
```

Hoe deze twee voorbeelden toch elkaar lijken

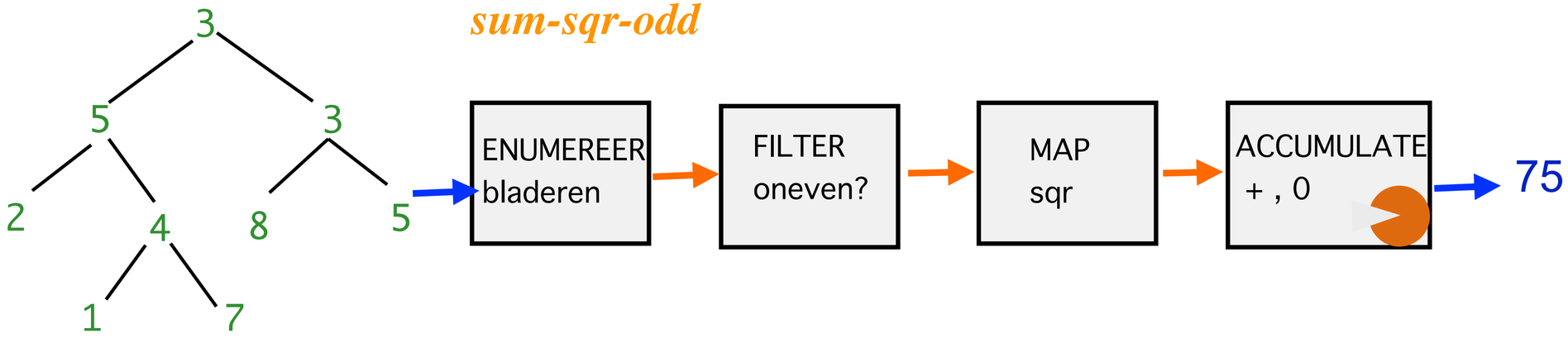
sum-sqr-odd

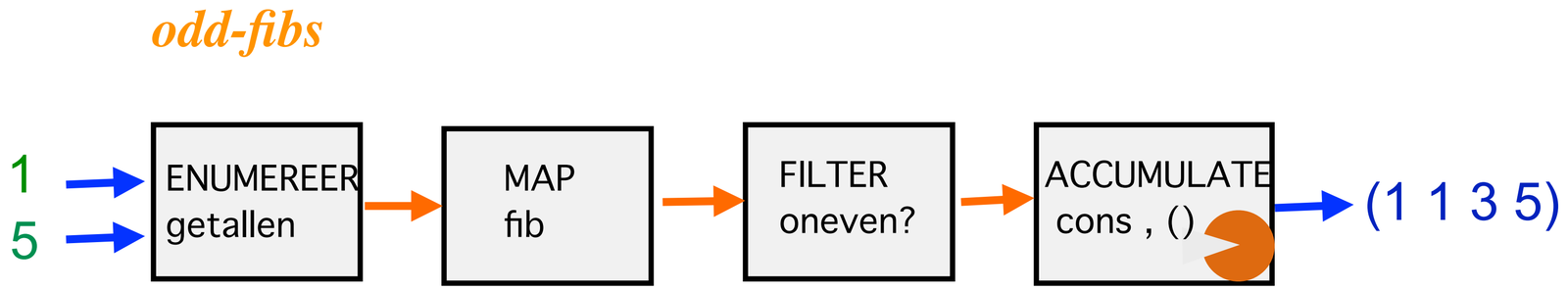
- enumereer de bladeren van een boom
- filter de oneven getallen er uit
- bereken het kwadraat van elk van die getallen
- accumuleer de resultaten, gebruik plus, start met 0

odd-fibs

- enumereer de gehele getallen tussen 1 en n
- bereken het fibonacci getal van elk van die getallen
- filter de oneven getallen er uit
- accumuleer de resultaten, gebruik cons, start met de lege lijst

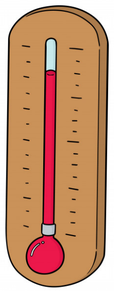
Deze twee voorbeelden in 'stream- programming'





Stromen als lijsten

```
(define the-empty-stream '())

(define (cons-stream el stream) (cons el stream))

(define (head stream) (car stream))

(define (tail stream) (cdr stream))

(define (empty-stream? stream) (null? stream))
```



21 20 13 24 19 21 23 18 15 17 20 22

(22 20 17 15 18 23 21 19 24 13 20 21)

Individuele bouwstenen (generators)

```
(define (enumerate-tree tree)
  (if (leaf? tree)
      (cons-stream tree the-empty-stream)
      (append-streams (enumerate-tree (left tree))
                      (enumerate-tree (right tree)))))

(define (append-streams s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream (head s1) (append-streams (tail s1) s2))))
```

```
(define (enumerate-int low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (enumerate-int (+ 1 low) high))))
```

Individuele bouwstenen (verwerkers)

```
(define (filter-odd stream)
  (cond ((empty-stream? stream) the-empty-stream)
        ((odd? (head stream))
         (cons-stream (head stream) (filter-odd (tail stream))))
        (else (filter-odd (tail stream)))))
```

```
(define (map-sqr stream)
  (if (empty-stream? stream)
      the-empty-stream
      (cons-stream (square (head stream)) (map-sqr (tail stream)))))
```

```
(define (map-fib stream)
  (if (empty-stream? stream)
      the-empty-stream
      (cons-stream (fib (head stream)) (map-fib (tail stream)))))
```

Individuele bouwstenen (accumulatoren)

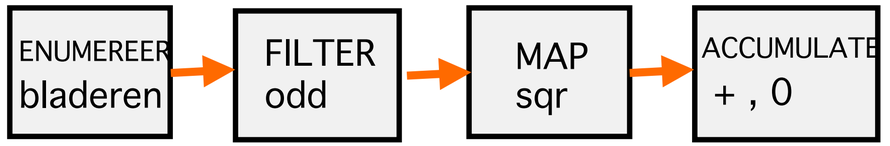
```
(define (accumulate-+ stream)
  (if (empty-stream? stream)
      0
      (+ (head stream) (accumulate-+ (tail stream)))))
```

```
(define (accumulate-cons stream)
  (if (empty-stream? stream)
      '()
      (cons (head stream) (accumulate-cons (tail stream)))))
```

is natuurlijk wat gek om van een stroom die een lijst is terug een lijst te maken maar volgende les zullen stromen geen gewone lijsten meer zijn

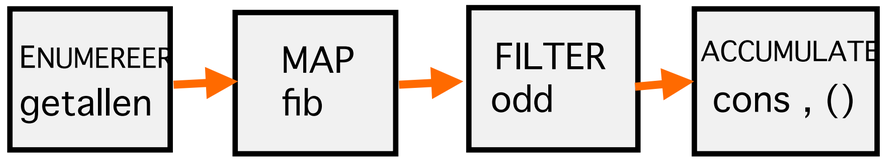
Individuele bouwstenen laten samenwerken (de voorbeelden)

```
(define (sum-sqr-odd tree)
  (accumulate-+
    (map-sqr
      (filter-odd
        (enumerate-tree tree)))))
```




```
> (sum-sqr-odd test)
75
```

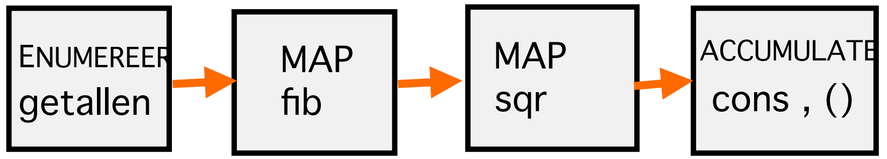
```
(define (odd-fibs n)
  (accumulate-cons
    (filter-odd
      (map-fib
        (enumerate-int 1 n)))))
```




```
> (odd-fibs 5)
(1 1 3 5)
```

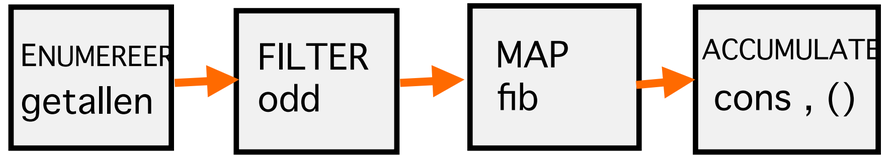
Individuele bouwstenen laten samenwerken (meer voorbeelden)

```
(define (list-square-fibs n)
  (accumulate-cons
    (map-sqr
      (map-fib
        (enumerate-int 1 n)))))
```




```
> (list-square-fibs 5)
(1 1 4 9 25)
```

```
(define (fibs-of-odd n)
  (accumulate-cons
    (map-fib
      (filter-odd
        (enumerate-int 1 n)))))
```

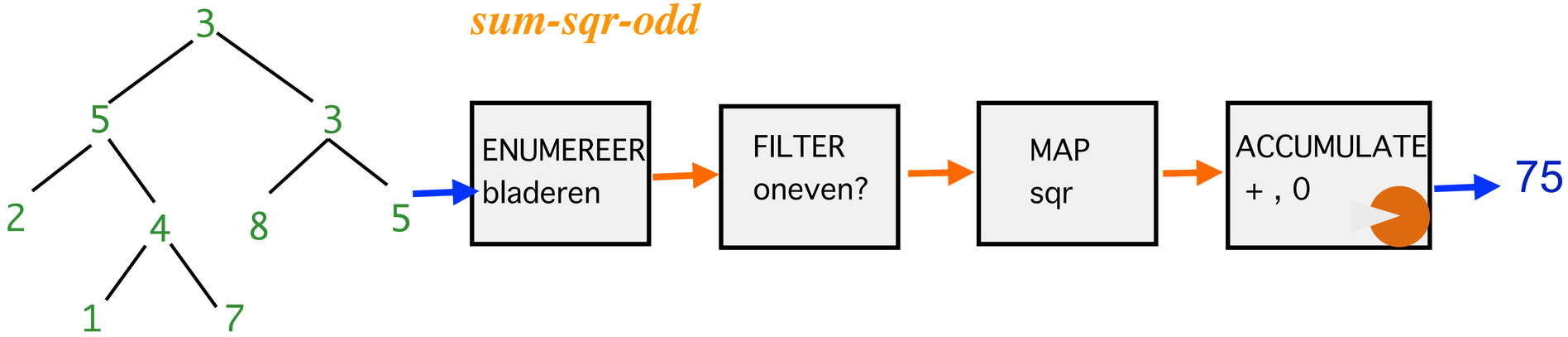


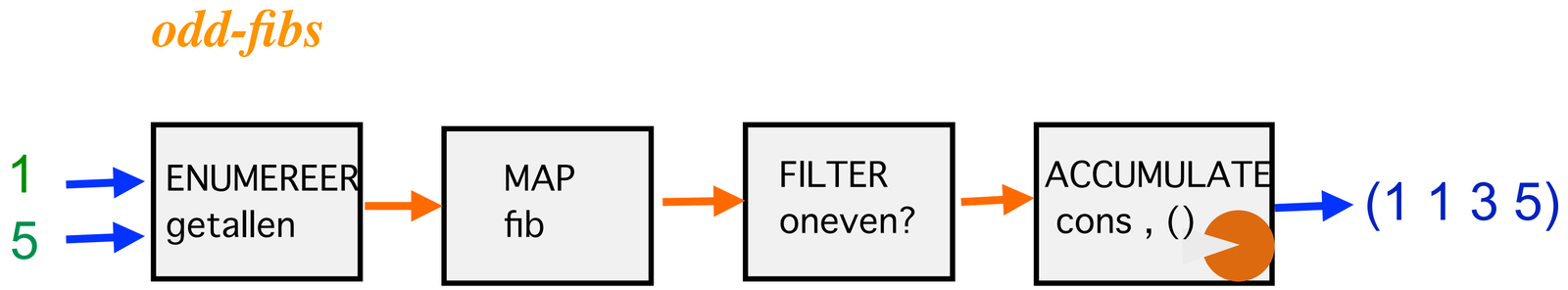

```
> (fibs-of-odd 5)
(1 2 5)
```

Les 12: Stromen

Sessie 2

Deze twee voorbeelden in 'stream- programming'

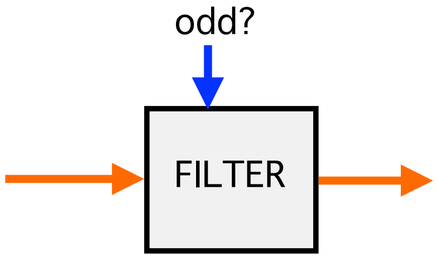




Hogere orde bouwstenen - Filter

```
(define (filter pred stream)
  (cond ((empty-stream? stream)
         the-empty-stream)
        ((pred (head stream))
         (cons-stream (head stream) (filter pred (tail stream))))
        (else (filter pred (tail stream)))))
```

```
> (filter odd? '(1 2 3 4 5))
(1 3 5)
> (filter even? '(1 2 3 4 5))
(2 4)
```

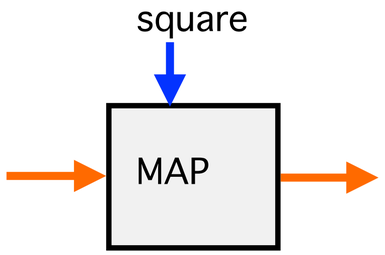


Hogere orde bouwstenen - MAP

```
(define (stream-map func stream)
   (if (empty-stream? stream)
       the-empty-stream
       (cons-stream
          (func (head stream))
          (stream-map func (tail stream)))))
```

mag niet de scheme map overschrijven, dus ik gebruik stream-map

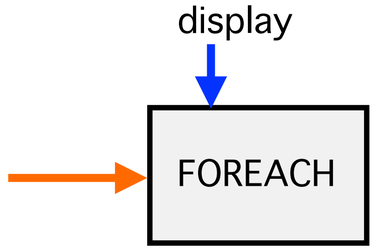
```
> (stream-map square '(1 2 3 4 5))
(1 4 9 16 25)
> (stream-map fac '(1 2 3 4 5))
(1 2 6 24 120)
```



Hogere orde bouwstenen - Foreach

```
(define (stream-for-each proc stream)
   (if (empty-stream? stream)
       'done
       (begin (proc (head stream))
                 (stream-for-each proc (tail stream)))))
```

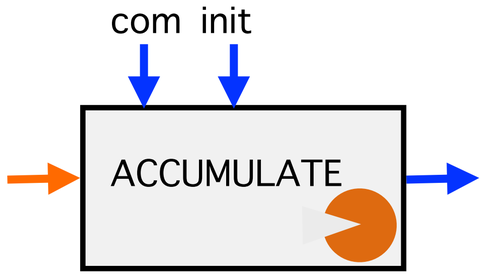
```
> (stream-for-each display '(1 2 3 4 5))
12345done
```



Hogere orde bouwstenen - Accumulate

```
(define (accumulate com init stream)
   (if (empty-stream? stream)
       init
       (com (head stream)
            (accumulate com init (tail stream)))))
```

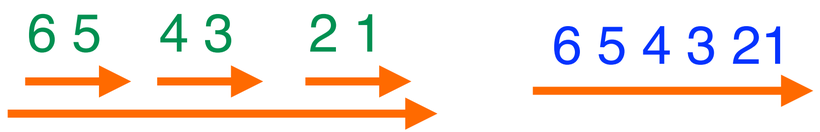
```
> (accumulate + 0 '(1 2 3 4 5))
15
> (accumulate * 1 '(1 2 3 4 5))
120
> (accumulate append '() '((1 2) (3 4) (5 6)))
(1 2 3 4 5 6)
```



Gebruik van hogere orde bouwstenen

```
(define (sum-stream stream)
  (accumulate + 0 stream))

(define (product-stream stream)
  (accumulate * 1 stream))

(define (make-list stream)
  (accumulate cons '() stream))

(define (flatten stream)
  (accumulate append-streams the-empty-stream stream))
```

```
> (flatten '((1 2) (3 4) (5 6)))
(1 2 3 4 5 6)
```

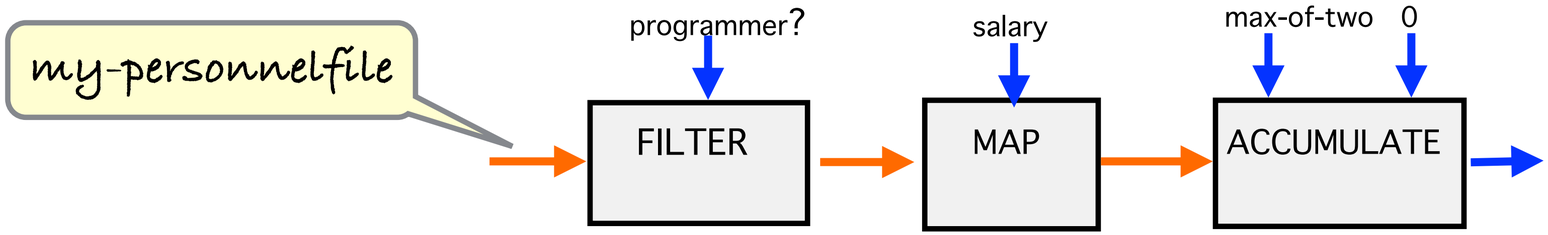


Conventionele dataverwerking met stromen (1)

```
(define my-personnelfile
  '((jan programmer 120000)
    (katy boss 150000)
    (an programmer 110000)
    (jef programmer 80000)
    (piet secretary 80000)))

(define (name record) (car record))
(define (position record) (cadr record))
(define (salary record) (caddr record))
```

Conventionele dataverwerking met stromen (2)

```
(define (highest-pay-of-programmer personnelfile)
  (accumulate max-of-two 0
    (stream-map salary
      (filter programmer? personnelfile))))

(define (programmer? record)
  (eq? 'programmer (position record)))

(define (max-of-two x y)
  (if (> x y) x y))
```




```
> (highest-pay-of-programmer my-personnelfile)
120000
```

Gebruik van lambda in programmeren met stromen

```
(define (plus-one stream)
  (stream-map (lambda (x) (+ x 1)) stream))

(define (plus-two stream)
  (stream-map (lambda (x) (+ x 2)) stream))

(define (plus-constant constant stream)
  (stream-map (lambda (x) (+ constant x)) stream))
```

de formele parameter van de omringende functiedefinitie kan gebruikt worden in de body van de lambda

Geneste mappings (fout)

```
(define (all-pairs-try low high)
  (stream-map (lambda (i)
                (stream-map (lambda (j)
                              (list i j))
                            (enumerate-int low high)))
              (enumerate-int low high)))
```

```
> (all-pairs-try 1 5)
(((1 1) (1 2) (1 3) (1 4) (1 5))
((2 1) (2 2) (2 3) (2 4) (2 5))
((3 1) (3 2) (3 3) (3 4) (3 5))
((4 1) (4 2) (4 3) (4 4) (4 5))
((5 1) (5 2) (5 3) (5 4) (5 5)))
```

too many brackets

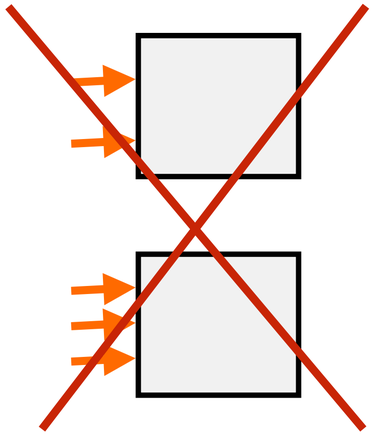
Geneste mappings (juist)

```
(define (all-pairs low high)
  (flatten
    (stream-map (lambda (i)
                  (stream-map (lambda (j)
                                (list i j))
                              (enumerate-int low high)))
                (enumerate-int low high))))
```

```
> (all-pairs 1 5)
((1 1) (1 2) (1 3) (1 4) (1 5)
(2 1) (2 2) (2 3) (2 4) (2 5)
(3 1) (3 2) (3 3) (3 4) (3 5)
(4 1) (4 2) (4 3) (4 4) (4 5)
(5 1) (5 2) (5 3) (5 4) (5 5))
```

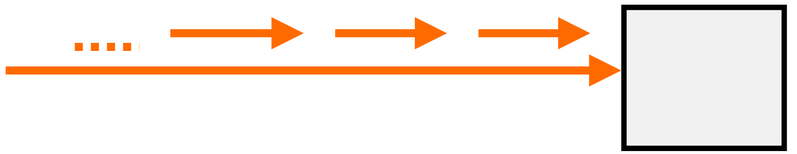
Accumuleren over een stroom van stromen

```
(define (acc-n com init streams)
  (if (empty-stream? (head streams))
      the-empty-stream
      (cons-stream
        (accumulate com init (stream-map head streams))
        (acc-n com init (stream-map tail streams)))))
```



```
> (acc-n + 0 '((1 2 3 4) (5 6 7 8) (1 1 1 1)))
(7 9 11 13)
> (acc-n + 0 '((1 2 3) (4 5 6) (7 8 9) (10 11 12)))
(22 26 30)
> (acc-n cons '() '((1 1 1) (2 2 2) (3 3 3)))
((1 2 3) (1 2 3) (1 2 3))
```

heel algemeen, onbeperkt aantal stromen op de inputstroom



Matrix algebra (1)

```
(define (dot-product vector1 vector2)
  (accumulate + 0
    (acc-n * 1
      (cons-stream vector1
        (cons-stream vector2
          the-empty-stream)))))
```

$\langle a_1\ a_2\ a_3 \rangle \otimes \langle b_1\ b_2\ b_3 \rangle =$
$a_1*b_1 + a_2*b_2 + a_3*b_3$

$b_3\ b_2\ b_1$ $\quad$ $a_3\ a_2\ a_1$

```
> (dot-product '(1 2 3) '(4 5 6))
32
```

```
(define (matrix-times-vector matrix vector)
  (stream-map (lambda (rij)
                (dot-product rij vector))
              matrix))
```

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \end{bmatrix} \otimes \langle v_1\ v_2\ v_3 \rangle$$

$m_{23}\ m_{22}\ m_{21}$ $\quad$ $m_{13}\ m_{12}\ m_{11}$ $\quad$ $v_3\ v_2\ v_1$

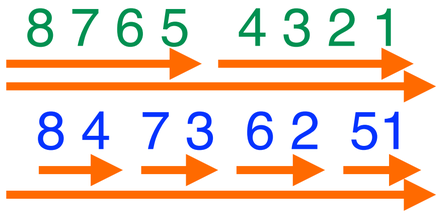
```
> (matrix-times-vector '((1 2 3 4) (5 6 7 8)) '(2 2 2 2))
(20 52)
```

Matrix algebra (2)

```
(define (transpose matrix)
  (acc-n cons-stream the-empty-stream matrix))
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

```
> (transpose '((1 2 3 4) (5 6 7 8)))
((1 5) (2 6) (3 7) (4 8))
```



$$\begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix}$$

```
(define (matrix-times-matrix matrix1 matrix2)
  (let ((cols (transpose matrix2)))
    (stream-map (lambda (row)
                  (matrix-times-vector cols row))
                matrix1)))
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix} \otimes \begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}$$

```
> (matrix-times-matrix '((1 2 3 4) (5 6 7 8)) '((1 2) (3 4) (5 6) (7 8)))
((50 60) (114 140))
```

$$\begin{bmatrix} 50 & 60 \\ 114 & 140 \end{bmatrix}$$

Permutaties

```
(define (permutations stream)
  (if (empty-stream? stream)
      (cons-stream the-empty-stream the-empty-stream)
      (flatten
        (stream-map (lambda (x)
                      (stream-map (lambda (y)
                                    (cons-stream x y))
                                  (permutations (remove x stream))))
                    stream))))

(define (remove x stream)
  (filter (lambda (y) (not (eq? x y))) stream))
```

```
> (permutations '(1 2 3))
((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))
```

Les 12: Stromen

Sessie 3

Stromen als delayed lists - motivatie

```
(define (sum-primes a b)
  (define (iter count result)
    (cond
      ((> count b) result)
      ((prime? count) (iter (+ count 1) (+ count result)))
      (else (iter (+ count 1) result))))
  (iter a 0))
```

iteratieve oplossing, geheugengebruik constant (en klein)

```
(define (sum-primes a b)
  (accumulate + 0 (filter prime? (enumerate-int a b))))
```

stream oplossing genereert altijd eerst een (lange) stroom (lijst)

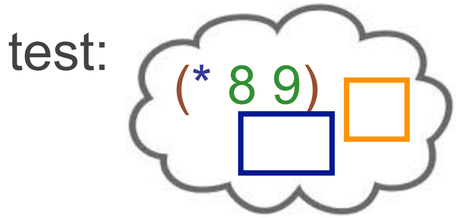
```
> (car (cdr (filter prime? (enumerate-int 2 1000000))))
```

zal tweede priemgetal in interval vinden maar wel eerst een lijst aanmaken van bijna een miljoen lang

Uitgestelde evaluatie

```
> (delay (* 2 3))
#<promise>

> (define test (delay (* 2 3)))

> test
#<promise>

> (force test)
6
```


```
> (#%require racket/promise)

> (define test (delay (* 8 9)))
> (promise? test)
#t
> (promise-forced? test)
#f
> (force test)
72
> (promise-forced? test)
#t
> (force test)
72
```

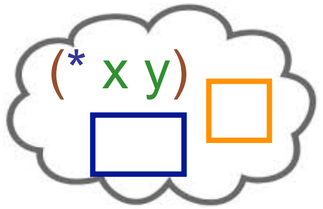


Uitgestelde evaluatie en variabelen

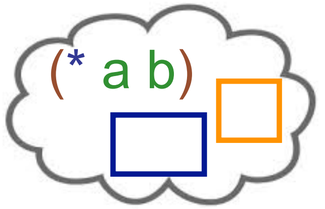
```
> (define x 2)
> (define y 3)

> (define test (delay (* x y)))
> (force test)
6

> (set! x 5)
> (force test)
6
```

```
> (define a 2)
> (define b 3)

> (define test (delay (* a b)))
> (set! a 5)
> (force test)
15

> (set! b 7)
> (force test)
15
```

test:



test:



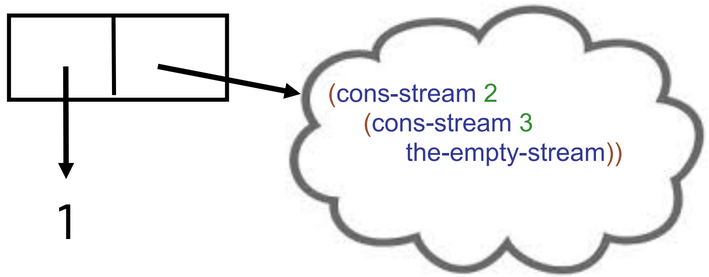
eens een promise geforceerd is wordt de waarde opgeslagen (cached) en bij een latere force wordt gewoon diezelfde waarde teruggegeven

Uitgestelde evaluatie om stromen te implementeren

```
(define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream el str)
     (cons el (delay str)))))

(define (tail stream) (force (cdr stream)))
```

wordt in les 14 uitgelegd hoe je zo een macro kan schrijven

```
> (define test (cons-stream 1
                            (cons-stream 2
                                (cons-stream 3 the-empty-stream))))
> test
(1 . #<promise>)
> (tail test)
(2 . #<promise>)
> (tail (tail test))
(3 . #<promise>)
```

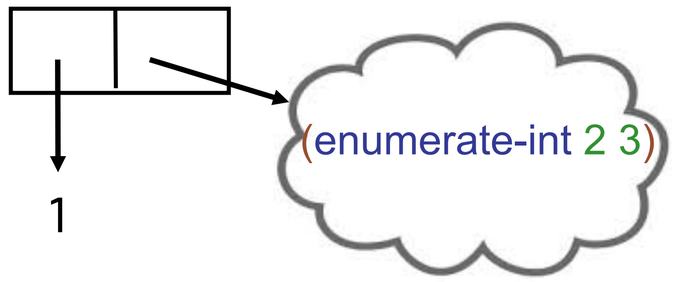


De generator herbekeken

```
(define (enumerate-int low high)
   (if (> low high)
       the-empty-stream
       (cons-stream low (enumerate-int (+ 1 low) high))))
```

code van slide 8 blijft onveranderd
werken met de nieuwe abstractie

```
> (define test (enumerate-int 1 3))
> test
(1 . #<promise>)
> (tail test)
(2 . #<promise>)
> (tail (tail test))
(3 . #<promise>)
```

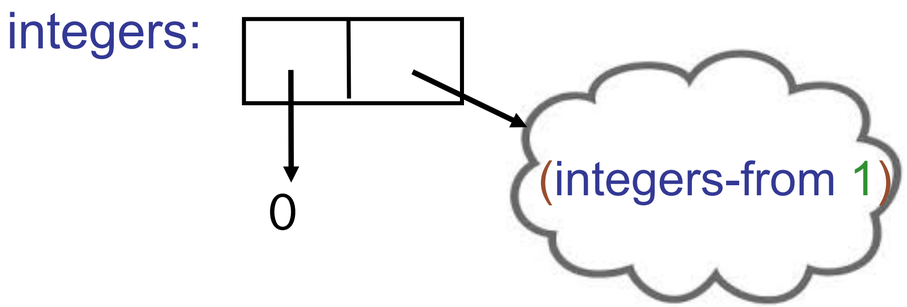


'Oneindig' lange stromen

```
(define (integers-from n)
  (cons-stream n (integers-from (+ 1 n))))

(define integers (integers-from 0))
```

constructieve recursie zonder eindconditie; omdat cons-stream 'lui' is kan dit gewoon

```
> integers
(0 . #<promise>)
```



'Oneindig' lange stromen bewerken

```
(define (show stream n)
  (define (loop i)
    (if (> i 0)
        (begin (display (head stream))
               (display "-")
               (show (tail stream) (- i 1)))))
  (loop n))
```

hulpfunctie om eerste n elementen van een stroom op het scherm te tonen

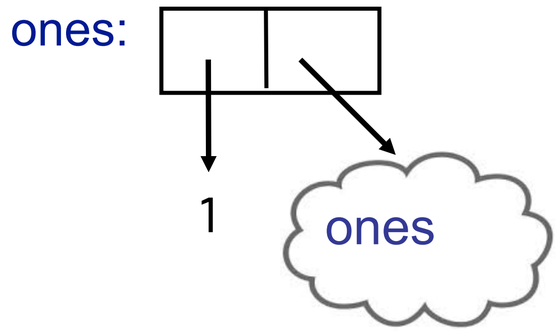
```
> (show integers 8)
0-1-2-3-4-5-6-7-
> (show (filter even? integers) 20)
0-2-4-6-8-10-12-14-16-18-20-22-24-26-28-30-32-34-36-38-
> (sum-stream integers)
Interactions disabled
```

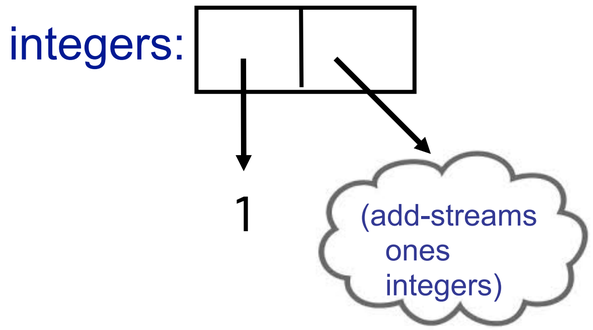
filter blijft gewoon werken

alle elementen van de oneindig lange stroom van integers optellen is natuurlijk een probleem

Stromen impliciet definiëren

```
(define ones (cons-stream 1 ones))

(define integers (cons-stream 1
                              (add-streams ones integers)))

(define (add-streams s1 s2)
    (cons-stream
        (+ (head s1) (head s2))
        (add-streams (tail s1) (tail s2))))
```

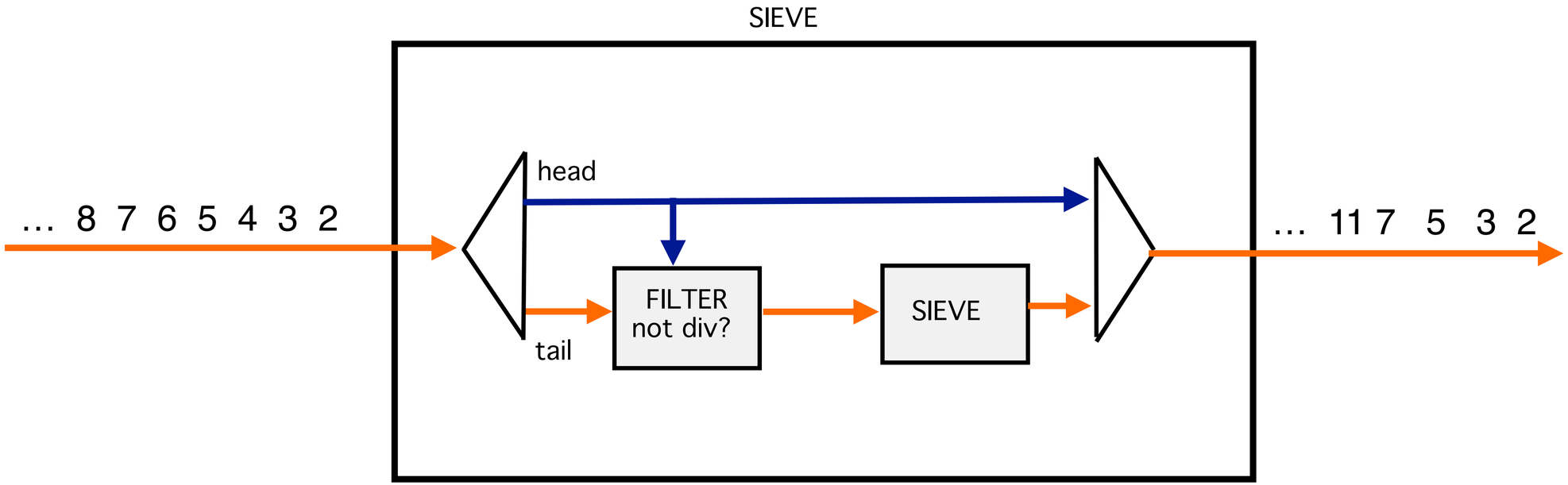





```
> (show ones 8)
1-1-1-1-1-1-1-1-
> (show integers 8)
1-2-3-4-5-6-7-8-
```

Priemgetallen: de zeef van Erastosthenes

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
  3   5   7       11   13   15   17   19   21   23   25
      5   7       11   13        17   19        23   25
          7       11   13        17   19        23
```



De zeef van Erastosthenes - stream implementatie

```
(define (div? x y)
  (= 0 (remainder x y)))

(define (sieve stream)
   (cons-stream
      (head stream)
      (sieve
         (filter
            (lambda (x) (not (div? x (head stream))))
            (tail stream)))))

(define primes
   (sieve (integers-from 2)))
```

```
> primes
(2 . #<promise>)
> (show primes 12)
2-3-5-7-11-13-17-19-23-29-31-37
```

Les 12: Stromen

Sessie 4

Geneste mappings over oneindige stromen (eerste poging)

```
(define (all-pairs low high)
  (flatten
    (stream-map (lambda (i)
                  (stream-map (lambda (j)
                                (list i j))
                              (enumerate-int low high)))
                (enumerate-int low high))))
```

code van slide 24 voor alle paren integers tussen low en high

```
(define (pairs s1 s2)
  (flatten
    (stream-map (lambda (i)
                  (stream-map (lambda (j)
                                (list i j))
                              s2))
                s1)))
```

zelfde idee voor alle paren elementen op 2 stromen

Geneste mappings over oneindige stromen (wat loopt fout?)

```
> (define test (enumerate-int 1 3))
> (show test 3)
1-2-3-
> (pairs test test)
((1 1) . #<promise>)
> (show (pairs test test) 9)
(1 1)-(1 2)-(1 3)-(2 1)-(2 2)-(2 3)-(3 1)-(3 2)-(3 3)-

> integers
(0 . #<promise>)
> (show integers 8)
0-1-2-3-4-5-6-7-
> (pairs integers integers)
Interactions disabled
```

ziet er best ok uit

bij nader inzien kan dit raar zijn voor oneindig lange stroom omdat alle paren met 1 zullen beginnen

de resulterende stroom is niet 'lui'

Geneste mappings over oneindige stromen (interleave idee)

de flatten diende om deelresultaten te combineren (slide 19)

```
(define (flatten stream)
  (accumulate
      append-streams
      the-empty-stream
      stream))
```

```
(define (flatten stream)
  (accumulate
      interleave
      the-empty-stream
      stream))
```

twee stromen combineren kan ook door om de beurt van elk een element te nemen

```
(define (append-streams s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream
          (head s1)
          (append-streams (tail s1) s2))))
```

```
(define (interleave s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream
          (head s1)
          (interleave s2 (tail s1)))))
```

Geneste mappings over oneindige stromen (wat loopt er nog mis?)

```
> test
(1 . #<promise>)
> (show test 3)
1-2-3-
> (pairs test test)
((1 1) . #<promise>)
> (show (pairs test test) 9)
(1 1)-(2 1)-(1 2)-(3 1)-(1 3)-(2 2)-(3 2)-(2 3)-(3 3)-

> integers
(0 . #<promise>)
> (show integers 8)
0-1-2-3-4-5-6-7-
> (pairs integers integers)
Interactions disabled
```

er komt nu een mix vooraan in de stroom

de resulterende stroom is nog altijd niet 'lui'

Geneste mappings over oneindige stromen (delay introduceren)

```
(define (accumulate com init stream)
  (if (empty-stream? stream)
      init
      (com (head stream)
           (accumulate com init (tail stream)))))
```

als com wordt gebonden aan en gewone operator gaat die de deelexpressies die argumenten zijn direct evalueren

```
(define (accumulate-delayed com init stream)
  (if (empty-stream? stream)
      init
      (com (head stream)
           (delay
             (accumulate-delayed com init (tail stream))))))
```

expliciet een delay toevoegen in accumulate

Geneste mappings over oneindige stromen (interleave aanpassen)

```
(define (interleave s1 s2)
  (if (empty-stream? s1)
      s2
      (cons-stream
         (head s1)
         (interleave s2 (tail s1)))))
```

maar deze interleave verwacht een stroom als tweede argument en niet een promise

```
(define (interleave-delayed s1 s2)
  (if (empty-stream? s1)
      (force s2)
      (cons-stream
         (head s1)
         (interleave-delayed (force s2) (delay (tail s1))))))
```

als tweede argument wordt een promise verwacht/gemaakt

Geneste mappings over oneindige stromen (en nu?)

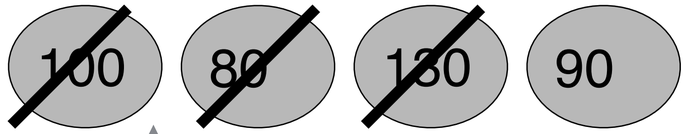
```
(define (flatten stream)
  (accumulate-delayed
      interleave-delayed
      the-empty-stream
      stream))
```

```
> integers
(0 . #<promise>)
> (show integers 8)
0-1-2-3-4-5-6-7-
> (pairs integers integers)
((0 0) . #<promise>)
> (show (pairs integers integers) 9)
(0 0)-(1 0)-(0 1)-(2 0)-(0 2)-(1 1)-(0 3)-(0 4)-(1 2)-(0 5)-(2 1)-
```

De simpele bankrekening van les 10

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "unknown request
                       —MAKE-ACCOUNT" m))))
  dispatch)
```

```
> (define viv (make-account 100))
> viv
#<procedure:dispatch>
> ((viv 'withdraw) 20)
80
> ((viv 'deposit) 50)
130
> ((viv 'withdraw) 40)
90
```



de balance wordt als lokale toestand opgeslagen telkens een transactie plaatsgrijpt

Een stroom die dezelfde informatie heeft; alles is functioneel, geen assignment nodig

```
(define (stream-account balance transactions)
  (cons-stream balance
     (stream-account
        (+ balance (head transactions))
        (tail transactions)))))
```

```
(define vivs-transactions
   (cons-stream -20
      (cons-stream +50
         (cons-stream -40
            the-empty-stream)))))

(define viv
 (stream-account 100 vivs-transactions))
```

de stroom bevat de opeenvolgende balances

```
> viv
(100 . #<promise>)
> (head vivs-account)
100
> (head (tail vivs-account))
80
> (head (tail (tail vivs-account)))
130
> (head (tail (tail (tail vivs-account))))
90
```

Les 13: Constraints