

Hoofdstuk 0,1

Nog wat Scheme die nuttig zal zijn



Inhoud

1. *De Named let*

2. *Vectoren*

3. *Libraries*

4. *Records*

De Named let

Algemene
Vorm

```
(let let-naam
  ((var1 exp1)
   (var2 exp2)
   . . .
   (varn expn))

. . .

(let-naam exp1' exp2' . . . expn')

. . .)
```

*Deze let wordt als de gewone let uitgevoerd,
maar kan in de body her-uitgevoerd worden
telkens met “verse” waarden voor de variabelen.
Hierdoor ontstaat er **een lus**.*

De Named let: Voorbeelden

```
(define (fac n)
  (let fac-loop
    ((result 1)
     (factor n))
    (if (= factor 0)
        result
        (fac-loop (* result factor) (- factor 1)))))
```



```
(define (fac n)
  (define (fac-loop result factor)
    (if (= factor 0)
        result
        (fac-loop (* result factor) (- factor 1))))
  (fac-loop 1 n))
```

```
(define (reverse l)
  (let rev-loop
    ((lst l)
     (acc '()))
    (if (null? lst)
        acc
        (rev-loop (cdr lst) (cons (car lst) acc)))))
```



```
(define (reverse l)
  (define (reverse-iter lst acc)
    (if (null? lst)
        acc
        (reverse-iter (cdr lst) (cons (car lst) acc))))
  (reverse-iter l '()))
```

Inhoud

1. *De Named let*

2. *Vectoren*

3. *Libraries*

4. *Records*

Vectoren

Een *vector* van grootte n bestaat uit n geheugenlocaties die in de computer *naast mekaar* opgeslagen worden. Je kan over cons-cellen nadenken als vectoren van grootte 2.

Verschillende Constructoren

```
> (make-vector 10)
#(0 0 0 0 0 0 0 0 0 0)
> (make-vector 10 "hahaha")
#{ "hahaha"
  "hahaha"
  "hahaha"
  "hahaha"
  "hahaha"
  "hahaha"
  "hahaha"
  "hahaha"
  "hahaha"
  "hahaha" }
> (vector "vanna" "een" "twee" "drie")
#("vanna" "een" "twee" "drie")
> (vector? (make-vector 10 "haha"))
#t
> (vector? (vector "vanna" "een" "twee" "drie"))
#t
> (pair? (vector "vanna" "een" "twee" "drie"))
#f
> (vector? (cons 1 2))
#f
> (define v0 (vector 1 "twee" "trois" 'четыри))
> v0
#(1 "twee" "trois" четыре)
> (vector-length v0)
4
```

Vectoren en
pairs zijn **echt**
verschillend



Procedures om vectoren te manipuleren

```
> (define v1 (vector 1 "twee" #\3 'четыри))
> (vector-ref v1 0)
1
> (vector-ref v1 3)
четыри
> (vector-ref v1 -1)
✖ ✖ vector-ref: contract violation
  expected: exact-nonnegative-integer?
  given: -1
  argument position: 2nd
  first argument...:
> (vector-ref v1 4)
✖ ✖ vector-ref: index is out of range
  index: 4
  valid range: [0, 3]
  vector: '#(1 "twee" #\3 четыре)
> (vector-set! v1 1 "dos cervezas por favor")
> v1
#(1 "dos cervezas por favor" #\3 четыре)
```



*Vectoren in Scheme zijn
inherent imperatieve/
destructieve datastructuren*

`(vector-ref vec idx)`

Accessor
(aka Getter)

Mutator (aka
Setter)


`(vector-set! vec idx val)`

Vectoren \neq Lijsten

```

> (define l (list 1 2 3))
> (set! l (cons 4 (cons 5 (cons 6 l))))
> l
(4 5 6 1 2 3)
> (list-ref l 2)
6
> (define v (vector 4 5 6 1 2 3))
> v
#(4 5 6 1 2 3)
> (vector-ref v 2)
6
> (define v-of-l (vector (list 1 2 3)
                          (list "un" "deux" "trois")
                          (list "one" "two" "three")
                          (list "раз" "два" "три")))
> (define l-of-v (list (vector 1 "un" "one" "раз")
                       (vector 2 "deux" "two" "два")
                       (vector 3 "trois" "three" "три")))
> (list-ref (vector-ref v-of-l 3) 2)
"три"
> (vector-ref (list-ref l-of-v 2) 3)
"три"

```



De lengte van een lijst ligt niet vast bij constructie. De lengte van een vector ligt vast bij constructie.

*list-ref is in $O(n)$
vector-ref is in $O(1)$*

Random
Access
Property

Gevalsstudie: Matrices

*Vectoren in Scheme zijn 1-dimensionaal.
Scheme heeft geen ingebouwde 'matrices'.*

*We gaan daarom aan het werk en
programmeren een infrastructuur bij elkaar
om matrices te kunnen bewerken.*

Hoeveel kolommen
heeft een matrix?

```
> (define myA (maak-matrix 3 2))
> (define myB (maak-matrix 2 4))
> myA
#(#(0 0) #(0 0) #(0 0))
> myB
#(#(0 0 0 0) #(0 0 0 0))
> (begin
  (ij! myA 0 0 1)
  (ij! myA 0 1 2)
  (ij! myA 1 0 3)
  (ij! myA 1 1 4)
  (ij! myA 2 0 5)
  (ij! myA 2 1 6)
  (ij! myB 0 0 1)
  (ij! myB 0 1 2)
  (ij! myB 0 2 3)
  (ij! myB 0 3 4)
  (ij! myB 1 0 5)
  (ij! myB 1 1 6)
  (ij! myB 1 2 7)
  (ij! myB 1 3 8))
> myA
#(#(1 2) #(3 4) #(5 6))
> myB
#(#(1 2 3 4) #(5 6 7 8))
> (rijen myA)
3
> (kolommen myA)
2
```

Creëer een
n x m matrix

Vul rij i en kolom j
op met een verse
waarde.

Hoeveel rijen
heeft een matrix?



Voorstellen van Matrices

```
(define (maak-matrix n m)
  (define rijen (make-vector n '()))
  (do ((rij 0 (+ rij 1)))
      ((= rij n) rijen)
      (vector-set! rijen rij (make-vector m 0))))
```

```
(define (rijen A)
  (vector-length A))
```

```
(define (kolommen A)
  (vector-length (vector-ref A 0)))
```

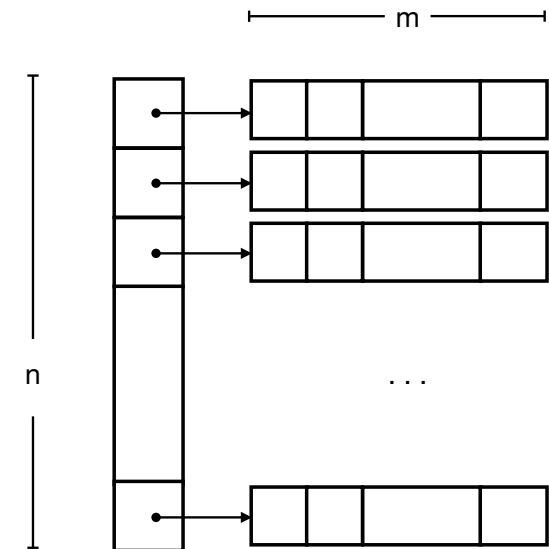
```
(define (ij? A i j)
  (vector-ref (vector-ref A i) j))
```

```
(define (ij! A i j v)
  (vector-set! (vector-ref A i) j v))
```

Welke waarde staat er op rij i en kolom j?

Verander waarde op rij i en kolom j

Een matrix *maken* van n rijen en m kolommen gebeurt door een vector te maken waarin we per rij een vector stoppen.



Bewerkingen op Matrices

```
(define (T A)
  (define AT (maak-matrix (kolommen A) (rijen A)))
  (do ((i 0 (+ i 1)))
      ((= i (rijen A)))
      (do ((j 0 (+ j 1)))
          ((= j (kolommen A)))
          (ij! AT j i (ij? A i j)))))
  AT)

(define (times A B)
  (define C (maak-matrix (rijen A) (kolommen B)))
  (do ((i 0 (+ i 1)))
      ((= i (rijen A)))
      (do ((j 0 (+ j 1)))
          ((= j (kolommen B)))
          (ij! C i j (let lus ((k 0)
                                (som 0))
                        (if (= k (kolommen A))
                            som
                            (lus (+ k 1) (+ som (* (ij? A i k) (ij? B k j)))))))))))
  C)
```

Maak de resulterende matrix en vul hem correct op: $AT_{ij} = A_{ji}$

Maak de resulterende matrix en vul hem correct op: $C_{ij} = \sum_k A_{ik} B_{kj}$



Inhoud

1. *De Named let*

2. *Vectoren*

3. *Libraries*

4. *Records*

Structuur van R7RS Code

```
#lang r7rs
```

```
(import (scheme base)
        (scheme write)
        (scheme read))
```

```
(define *patients* ...)
```

```
(define (nieuwe-patient)
  ...)
```

```
(define (dokter-beschikbaar)
  ...)
```

```
(define (loop)
  ...)
```

```
(loop)
```

Een **R7RS programma** is een file met een reeks imports van libraries, een reeks mogelijke top level definities en een expressie om het programma op te starten.

```
#lang r7rs
```

```
(define-library (kmp)
  (export match)
  (import (scheme base)
          (scheme write)))
```

```
(begin
  (define (compute-failure-function p)
    ...)

  (define (match t p)
    ...)))
```

Elke **R7RS library** bestaat uit een reeks definities, een reeks imports, een naam en een reeks exports

```
(import clause1 ... clausen)
```

Import is niet transitief



Botsingen

Name Clashes

```
(define-library (box1)
  (import (scheme base))
  (export f)

  (begin
    (define (f x y)
      (* x y)))))
```

```
(define-library (box2)
  (import (scheme base))
  (export f)

  (begin
    (define (f x)
      (* x x)))))
```

```
(import (a-d examples clash1)
  (a-d examples clash2))
```

```
✗ ✗ module: identifier already required
  also provided by: a-d/examples/clash1 in: function
>
```

Het kan voorkomen dat je 2 verschillende libraries wil gebruiken die ooit dezelfde namen gekozen hebben voor hun procedures.



Omgaan met Name Clashes

(import clause₁ ... clause_n)

"alles"

(dir₁ dir₂ dir_n file-name)

"enkel sommige procedures"

(only clause name₁ name₂ ... name_k)

plak een naam voor alle namen

(prefix clause name)

alles behalve sommige procedures

(except clause name₁ name₂ ... name_k)

hernoem sommige procedures

(rename clause (name₁ new-name₁)
(name₂ new-name₂)
...
(name_k new-name_k))

(import (scheme base)
(scheme write))

(import (only (scheme write) display write))

(import (prefix (scheme write) IO~=))

> (IO~=display "test")
test

(import (except (scheme base) define))

> (define x 10)
× × define: unbound identifier in: define

(import (a-d examples clash1)
(rename (a-d examples clash2) (f g)))

> (f 10 20)
200
> (g 10)
100



Inhoud

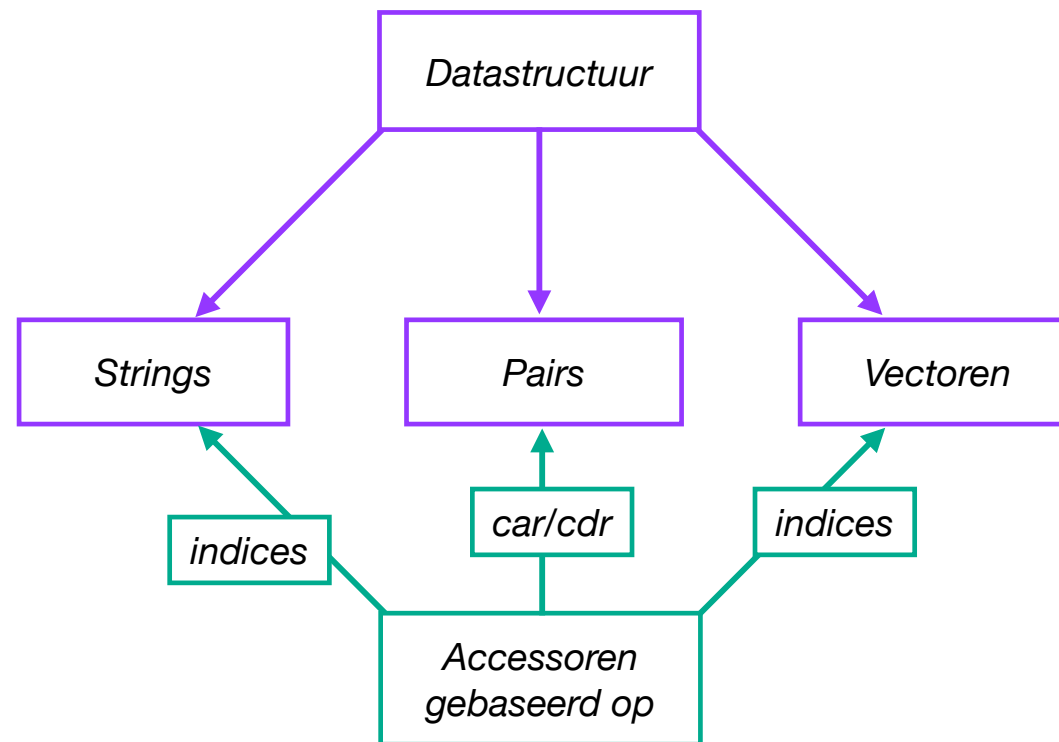
1. *De Named let*

2. *Vectoren*

3. *Libraries*

4. *Records*

“Datalijm” tot nu toe gezien in R5RS



Programmeeropdracht: Stel “een persoon” voor

Niet 1 persoon, maar
code om personen
voor te stellen en te
manipuleren.

*Een **persoon** heeft een naam, een voornaam, een leeftijd en een salaris. Enkel de leeftijd en salaris kunnen veranderen.*

Abstract Data Type

Elk ADT heeft
een naam

Iedere operatie heeft argumenten
van een zeker type en geeft
(eventueel) een resultaat weer
van een zeker type.

en een lijst van namen
van operaties die we op
de waarden van het
ADT kunnen toepassen

```
ADT person

new
  ( string string number number → person )
person?
  ( any → boolean )
name
  ( person → string )
surname
  ( person → string )
age
  ( person → number )
age!
  ( person number → person )
salary
  ( person → number )
salary!
  ( person number → person )
```

3 ≠ Procedurele Implementaties

```
(define-library (person)
  (export new person? name surname age age! salary salary!)
  (import (scheme base))
  (begin
```

Hier staat de code om personen te
“*representeren*” en code voor de
accessoren/mutatoren/operaties op
personen te “*implementeren*”

```
))
```



Voorbeeld van Gebruik

```
(import (a-d examples person-3)
        ;(a-d examples person-2)
        ;(a-d examples person-1)
        (rnrs base)
        (rnrs io simple))

(define jef      (new "Jef" "Vandenbrande" 30 40000))
(define anoushka (new "Anoushka" "Sheremetyevo" 31 50000))

(define (birthday! p)
  (age! p (+ 1 (age p))))

(define (earns-more? p1 p2)
  (> (salary p1) (salary p2)))

(define (communism! p1 p2)
  (define p1-sal (salary p1))
  (define p2-sal (salary p2))
  (define average (/ (+ p1-sal p2-sal) 2))
  (salary! p1 average)
  (salary! p2 average)
  average)
```

Bepaalt welke
implementatie van het
ADT we momenteel
gebruiken

```
> (earns-more? jef anoushka)
#f
> (communism! jef anoushka)
45000
> (salary jef)
45000
> (salary anoushka)
45000
> (age jef)
30
> (birthday! jef)
{person "Jef" "Vandenbrande" 31 45000}
> (age jef)
31
>
```



Implementatie #1: Met Pairs



```
(define-library (person)
  (export new person? name surname age age! salary salary!)
  (import (scheme base)
          (scheme cxx))
  (begin
    (define (new name snam age sal)
      (list 'person name snam age sal))

    (define (person? sval)
      (and (pair? sval) (eq? (car sval) 'person)))

    (define (name prsn)
      (cadr prsn))

    (define (surname prsn)
      (caddr prsn))

    (define (age prsn)
      (cddddr prsn))

    (define (age! prsn)
      (set-car! (cddddr prsn)))

    (define (salary prsn)
      (set-car! (cddddr prsn)))

    (define (salary! prsn slry)
      (set-car! (cddddr prsn) slry))))
```

Elke persoon wordt *voorgesteld* door een lijst met zijn/haar informatie erin. De lijst wordt getagged met het symbol 'person.

De *operaties* op personen worden *geïmplementeerd* met de traditionele cxx operaties van Scheme



Implementatie #2: Met vectoren



```
(define-library (person)
  (export new person? name surname age age! salary salary!)
  (import (scheme base))
  (begin
    (define person-tag 'person)
    (define (new n sn a s)
      (vector person-tag n sn a s))
    (define tag-index 0)
    (define nam-index 1)
    (define sur-index 2)
    (define age-index 3)
    (define sal-index 4)
    (define (person? x)
      (and (vector? x)
           (eq? (vector-ref x tag-index) person-tag)))
    (define (name p)
      (vector-ref p nam-index))
    (define (surname p)
      (vector-ref p sur-index))
    (define (age p)
      (vector-ref p age-index))
    (define (age! p a)
      (vector-set! p age-index a)
      p)
    (define (salary p)
      (vector-ref p sal-index))
    (define (salary! p s)
      (vector-set! p sal-index s)
      p)))
```

Elke persoon wordt *voorgesteld* door een vector met zijn/haar informatie erin. De eerste vector entry bevat het symbol 'person.

De *operaties* op personen worden *geïmplementeerd* met *vector-ref* en *vector-set!*



Implementatie #3: Met Records



```
(define-library (person)
  (export new person? name surname age age! salary salary!)
  (import (scheme base))
  (begin

    (define-record-type person
      (new n sn a s)
      person?
      (n name)
      (sn surname)
      (a age age!)
      (s salary salary!))))
```

*Elke persoon wordt **voorgesteld** door een record met zijn/haar informatie erin.*

*De **operaties** op personen worden automatisch gegenereerd door R7Rs.*



Record Types: Algemene Vorm

```
(define-record-type NAAM_VAN_TYPE  
  (CONSTRUCTOR veld1 veld2 ... veldn)  
  TYPE_TEST?  
  (veld1 accessor1 mutator1)  
  (veld2 accessor2 mutator2)  
  ...  
  (veldn accessorn mutatorn))
```

De mutators zijn
optioneel

Bij de definitie van een record type mag je alle namen van de procedures die gegenereerd zullen worden zelf kiezen. De naam van het type wordt gebruikt door de REPL om waarden van het type te printen. Elk veld moet minstens een accessor procedure hebben. Als je de naam van de mutator weglaat is het veld niet destructief aanpasbaar.

Voorbeeld 2

Triples of “trons-cellen” hebben een car, een cdr en een cmr.

ADT triple

```
trons
  ( any any any → triple )
triple?
  ( any → boolean )
car
  ( triple → any )
set-car!
  ( triple any → ∅ )
cdr
  ( triple → any )
set-cdr!
  ( triple any → ∅ )
cmr
  ( triple → any )
set-cmr!
  ( triple any → ∅ )
```

Implementatie: probeer het!

```
> (define t (trons 1 2 3))
> (pair? t)
#f
> (triple? t)
#t
> (car t)
1
> (cdr t)
3
> (cmr t)
2
> (set-car! t "hi")
> (set-cmr! t "di")
> (set-cdr! t "ho")
> t
#(struct:triple "hi" "di" "ho")
> (car t)
"hi"
> (cmr t)
"di"
> (cdr t)
"ho"
```



Hoofdstuk 0,1

1. De Named Let
2. Vectoren
3. Libraries
4. Records

