Les 4: Hogere orde procedures

SESSIE 1

Les 4: Hogere orde procedures

Alle procedures die we tot nu toe bestudeerden nemen getallen als parameters en produceren meestal ook een getal als resultaat. Later gaan we natuurlijk andere soorten data bestuderen en gaan we ook procedures schrijven die complexe data-objecten als parameter nemen en eventueel als waarde teruggeven.

In deze les gaat onze aandacht echter naar procedures die niet alleen data-objecten maar ook procedure-objecten kunnen manipuleren, m.a.w. procedures die procedures als parameter nemen en procedures die procedures opleveren als resultaat.

Procedures als eerste klasse objecten

Men noemt objecten in een programmeertaal 'eerste klasse objecten' wanneer ze:

- kunnen toegekend worden aan een variabele
- kunnen doorgegeven worden als argument aan een procedure
- kunnen teruggegeven worden als resultaat van een procedureoproep

Uit deze beschrijving volgt dat procedure-objecten net als data-objecten eerste klas objecten zijn in Scheme, iets wat in veel andere programmeertalen niet het geval is.

Overzicht

De eerste reeks voorbeelden illustreert hoe patronen die gemeenschappelijk zijn aan verschillende procedures kunnen vastgelegd worden in één enkele hogere orde procedure.

Tussendoor worden in deze les ook de concepten anonieme procedures en lokale variabelen geïntroduceerd.

De tweede reeks voorbeelden zijn standaardvoorbeelden voor hogere orde procedures, i.e. functies waarvan de functiewaarden in verschillende punten nodig zijn in een berekening kunnen gewoon parameter zijn van de procedure die de berekening implementeert.

Tenslotte volgen nog een paar voorbeelden waar de hogere orde procedure een andere procedure als resultaat teruggeeft.

Enkele voorbeelden

SUM-INTEGERS 5 9 $5 + 6 + 7 + 8 + 9$

SUM-SQUARES 5 9 $25 + 36 + 49 + 64 + 81$

SUM-EVERY-TWO-INTEGERS 5 9 $5 + 7 + 9$

PI-SUM 5 13 $\frac{1}{5*7} + \frac{1}{9*11} + \frac{1}{13*15}$

PI-SUM a b $\frac{1}{a*(a+2)} + \frac{1}{(a+4)*((a+4)+2)} + \frac{1}{(a+8)*((a+8)+2)} + \ldots$

voor a = 1 convergeert deze reeks naar PI/8

CODE

Procedures met eenzelfde patroon

```
(define (sum-ints a b)
  (if (> a b)
      0
      (+ a
         (sum-ints (+ a 1) b))))
```

```
> (sum-ints 1 7)
28
```

```
(define (sum-evr-two-ints a b)
  (if (> a b)
      0
      (+ a
         (sum-evr-two-ints (+ a 2) b))))
```

```
> (sum-evr-two-ints 1 7)
16
```

```
(define (sum-sqrs a b)
  (if (> a b)
      0
      (+ (square a)
         (sum-sqrs (+ 1 a) b))))
```

```
> (sum-sqrs 1 7)
140
```

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2)))
         (pi-sum (+ a 4) b))))
```

```
> (pi-sum 1 7)
0.3619047619047619
```

Een hogere orde procedure kan dat patroon 'vatten'

```
(define (sum a b term next)
   (if (> a b)
      0
      (+ (term a)
         (sum (next a) b term next ))))
```

$$\sum_{i=a}^{b} (\text{term } i) \qquad (\text{next } i)$$

```
(define (square x) (* x x))
(define (double x) (* 2 x))
(define (cube x) (* x x x))

(define (plus-one x) (+ 1 x))
(define (plus-two x) (+ 2 x))
(define (plus-four x) (+ 4 x))
```

CODE

```
> (sum 1 7 id plus-one)
28
> (sum 1 7 square plus-one)
140
> (sum 1 7 id plus-two)
16
> (sum 5 100 cube plus-four)
5998824
> (sum 1 2 cube plus-two)
1
```

Een procedure bouwen met een hogere orde procedure

```
(define (pi-sum a b)
   (define (pi-term x)
      (/ 1 (* x (+ x 2))))
   (define (pi-next x)
      (+ x 4))
   (sum a b pi-term pi-next))
```

```
> (* (pi-sum 1 100) 8.0)
3.1215946525910105
> (* (pi-sum 1 1000) 8.0)
3.1395926555897833
```

(Anonieme) procedures construeren met lambda

```
(define (pi-sum a b)
  (sum a b
    (lambda (x) (/ 1 (* x (+ x 2))))
    (lambda (x) (+ x 4))))
```

```
> (* (pi-sum 1 100) 8.0)
3.1215946525910105
> (* (pi-sum 1 1000) 8.0)
3.139592655897833
```

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum a b pi-term
               pi-next))
```

(lambda (<formal-parameters>) <body>)

CODE

Define herbekeken (1)

```
> (lambda (x) (* x 2))
#<procedure>

> ((lambda (x) (* x 2)) 5)
10

> (define double (lambda (x) (* x 2)))

> (double 5)
10

> (define (double (x) (* x 2)))

> (double 5)
10
```

Define herbekeken (2)

(define (<name> <formal-parameters>) <body>)

is syntactic sugar for

(define <name> <expression>)

combined with

(lambda (<formal-parameters>) <body>)

Les 4: Hogere orde procedures

SESSIE 2

Locale variabelen - Motivatie

$$f: x \rightarrow \frac{(x^2 + 1)}{(x^2 - 1)}$$

CODE

```
(define (f1 x)
   (/  (+ (square x) 1)
       (- (square x) 1)))
```

```
> (f1 2.0)
1.666666666666667
```

```
(define (f3 x)
   (define (f-help y) (/ (+ y 1) (- y 1)))
   (f-help (square x)))
```

```
> (f3 2.0)
1.666666666666667
```

```
(define (f2 x)
   (define y (square x))
   (/ (+ y 1) (- y 1)))
```

```
> (f2 2.0)
1.666666666666667
```

```
(define (f4 x)
   ((lambda (y) (/ (+ y 1) (- y 1)))
    (square x)))
```

```
> (f4 2.0)
1.666666666666667
```

Locale variabelen: let special form

```
(define (f x)
   (let ((y (square x)))
      (/ (+ y 1) (- y 1))))
```

```
> (f 2.0)
1.6666666666666667
```

```
(let ( (<var-1> <expr-1>)
       (<var-2> <expr-2>)
       . . .
       (<var-n> <expr-n>) )
 <body>)
```

let is syntactische suiker

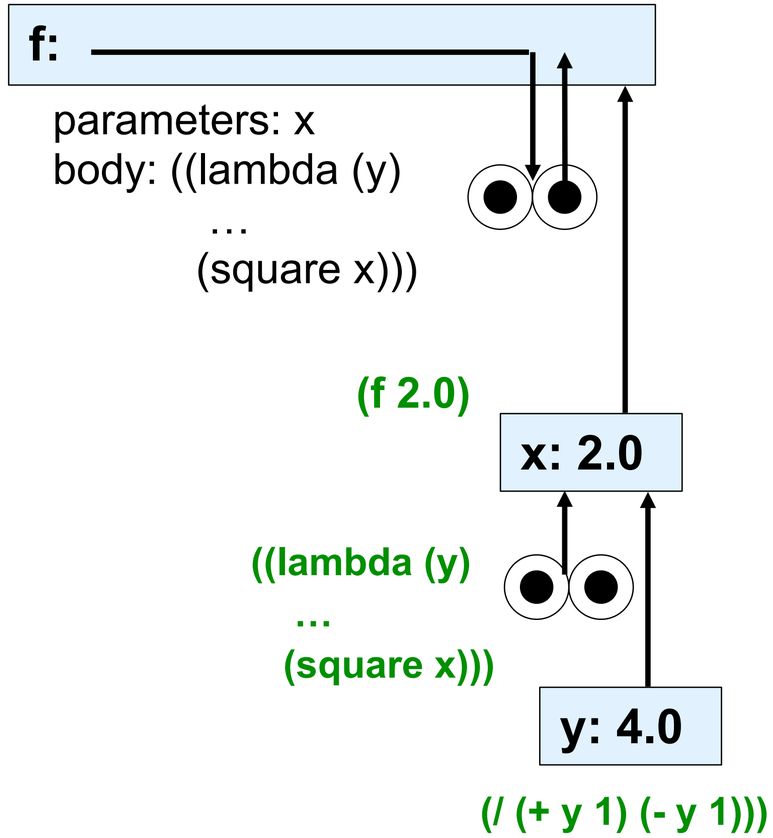
```
(let ((<var-1> <expr-1>)
      (<var-2> <expr-2>)
       . . .
      (<var-n> <expr-n>))
  <body>)
```

is syntactic sugar for

```
((lambda (<var-1>..<var-n>)
   <body>)
  <expr-1>
  . . .
  <expr-n>)
```

```
(define (f x)
  (let ((y (square x)))
    (/ (+ y 1) (- y 1))))
```

```
(define (f x)
  ((lambda (y)
     (/ (+ y 1) (- y 1)))
   (square x)))
```



Locale variabelen - let <> let*

```
(define (test1 a b)
  (let ((a 5)
        (c (+ a 2)))
    (display a)
    (display b)
    (display c)))
```

```
> (test1 1 2)
523
```

```
(define (test2 a b)
  (let* ((a 5)
         (c (+ a 2)))
    (display a)
    (display b)
    (display c)))
```

```
> (test2 1 2)
527
```

let

```
(let ((<var-1> <expr-1>)
      (<var-2> <expr-2>)
      . . .
      (<var-n> <expr-n>))
  <body>)
```

is syntactic sugar for

```
((lambda (<var-1>..<var-n>)
   <body>)
  <expr-1>
  . . .
  <expr-n>)
```

```
(define (test a b)
  (let ((a 5)
        (c (+ a 2)))
    (display a)
    (display b)
    (display c)))
```

```
(define (test a b)
  ((lambda (a c)
     (display a)
     (display b)
     (display c))
   5 (+ a 2)))
```

let*

```
(let* ((<var-1> <expr-1>)
       (<var-2> <expr-2>)
         . . .
       (<var-n> <expr-n>))
  <body>)
```

is syntactic sugar for

```
((lambda (<var-1>)
   ((lambda (<var-2>)
       . . .
      ((lambda (<var-n>)
          <body>)
          <expr-n>)
        . . .
    <expr-2>)
 <expr-1>)
```

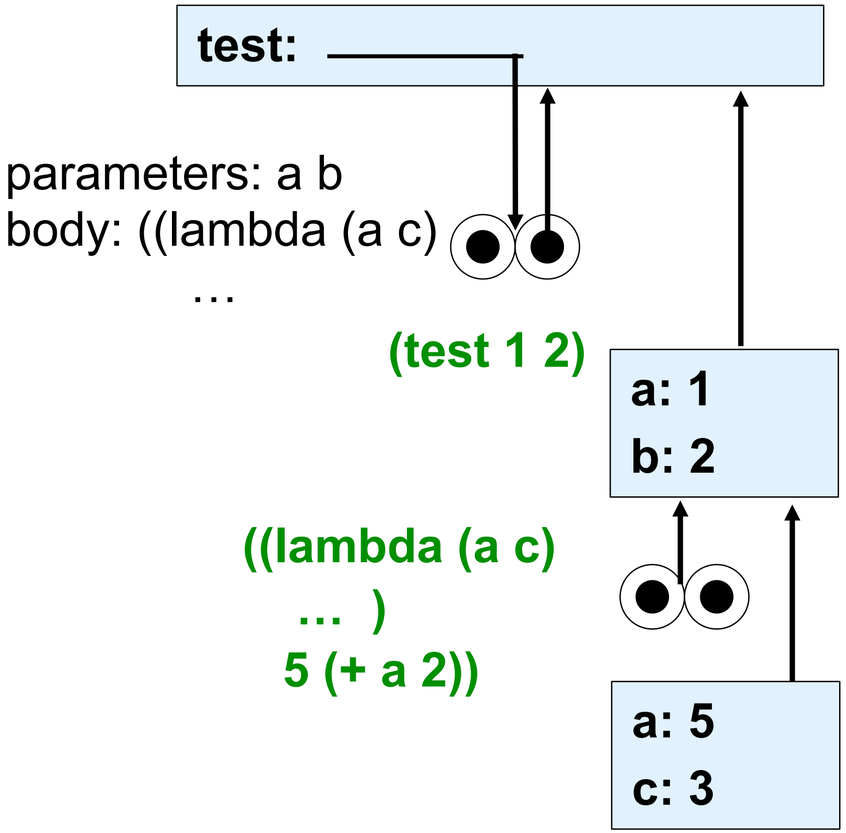
```
(define (test a b)
  (let* ((a 5)
         (c (+ a 2)))
    (display a)
    (display b)
    (display c)))
```

```
(define (test a b)
  ((lambda (a)
     ((lambda (c)
        (display a)
        (display b)
        (display c))
      (+ a 2)))
   5))
```

Locale variabelen - let

```
(define (test a b)
  ((lambda (a c)
     (display a)
     (display b)
     (display c))
   5 (+ a 2)))
```

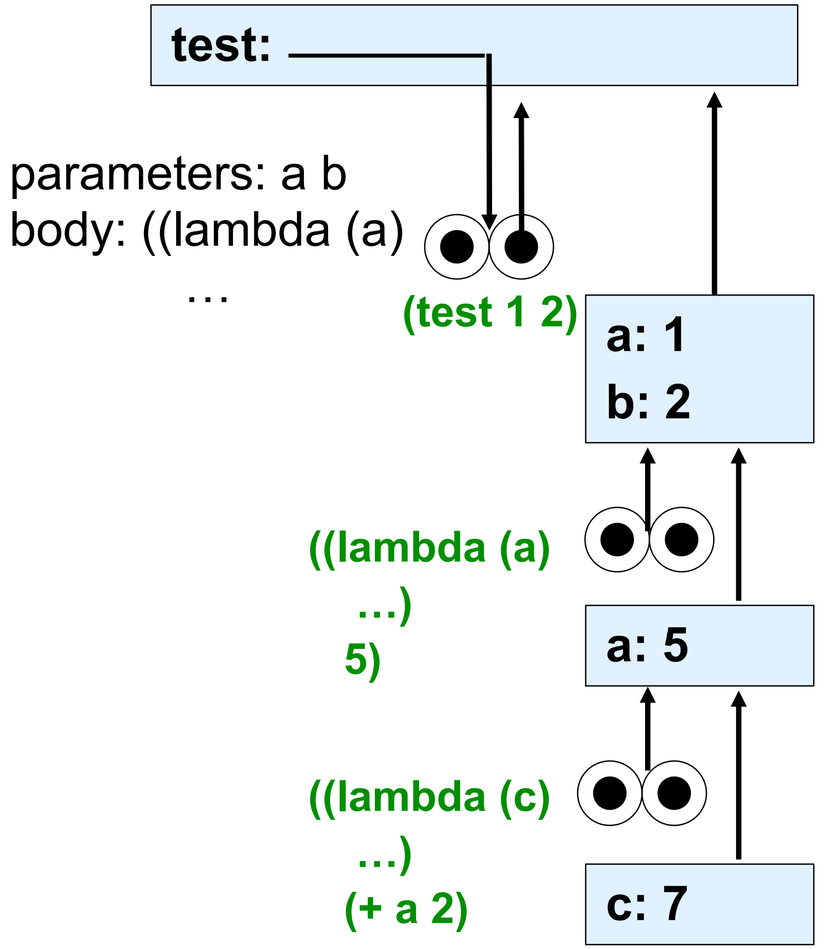
```
> (test 1 2)
523
```



Locale variabelen - let*

```
(define (test a b)
  ((lambda (a)
     ((lambda (c)
        (display a)
        (display b)
        (display c))
      (+ a 2)))
   5))
```

```
> (test 1 2)
527
```



Locale variabelen - letrec (1)

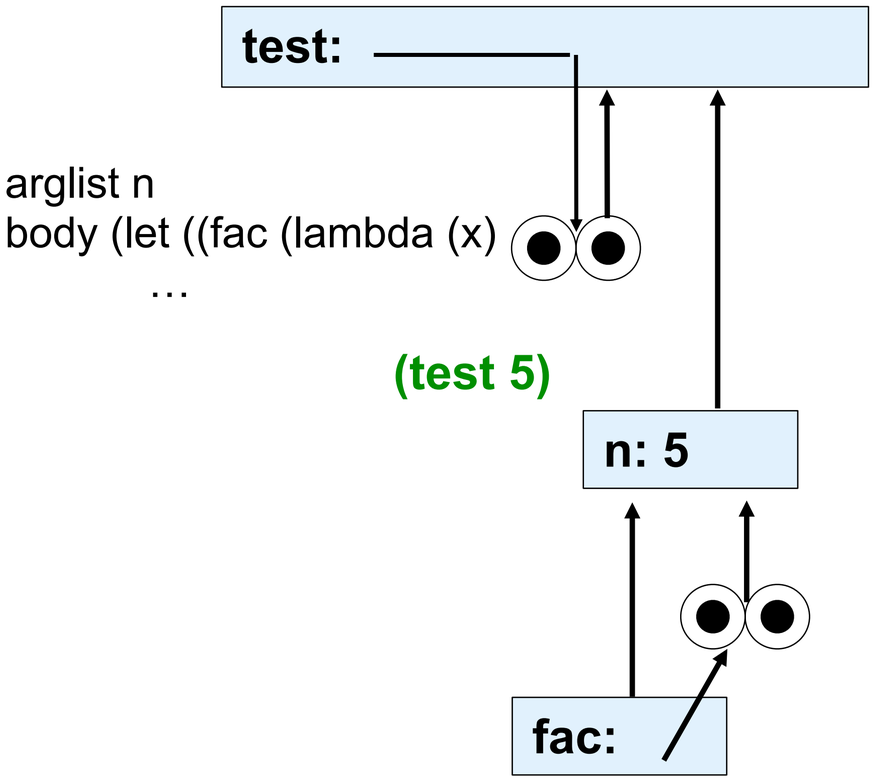
```
(define (test n)
  (let ((fac (lambda (x)
               (if (= x 0)
                   1
                   (* x (fac (- x 1)))))))
    (fac n)))
```

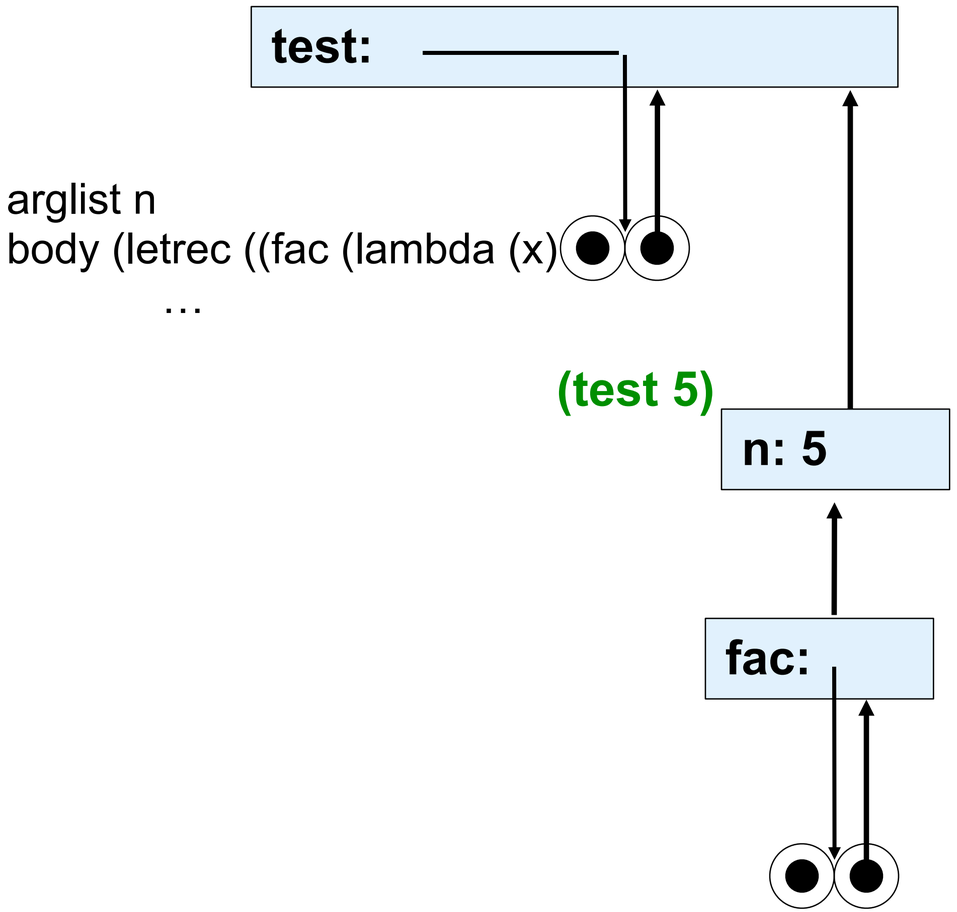
```
> (test 5)
⊗⊗ fac: undefined;
cannot reference undefined identifier
```

```
(define (test n)
  (letrec ((fac (lambda (x)
                  (if (= x 0)
                      1
                      (* x (fac (- x 1)))))))
    (fac n)))
```

```
> (test 5)
120
```

Locale variabelen - let <> letrec (2)

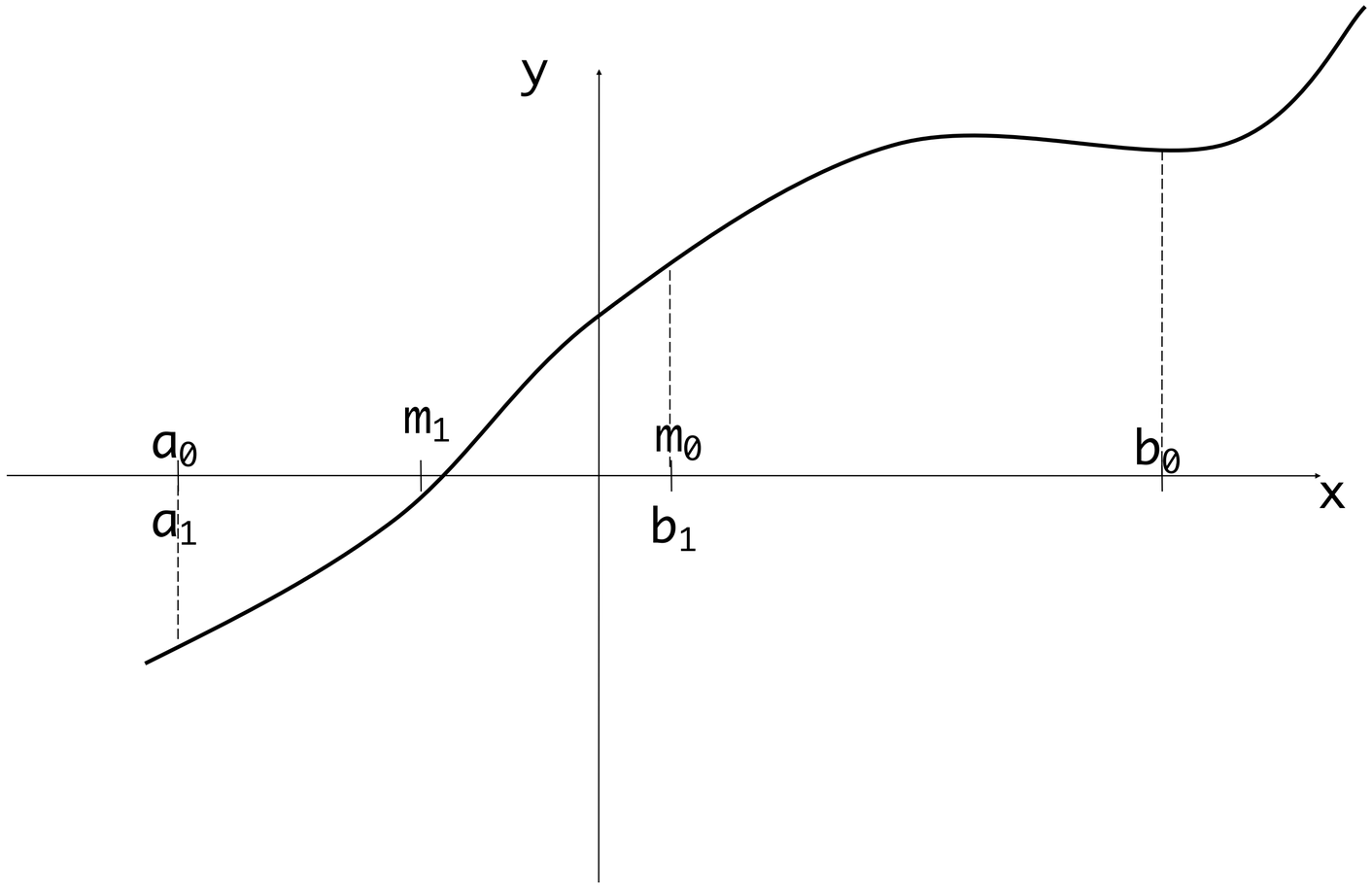




Les 4: Hogere orde procedures

SESSIE 3

Nulpunt van een functie benaderen met de half interval methode



De half interval methode

CODE

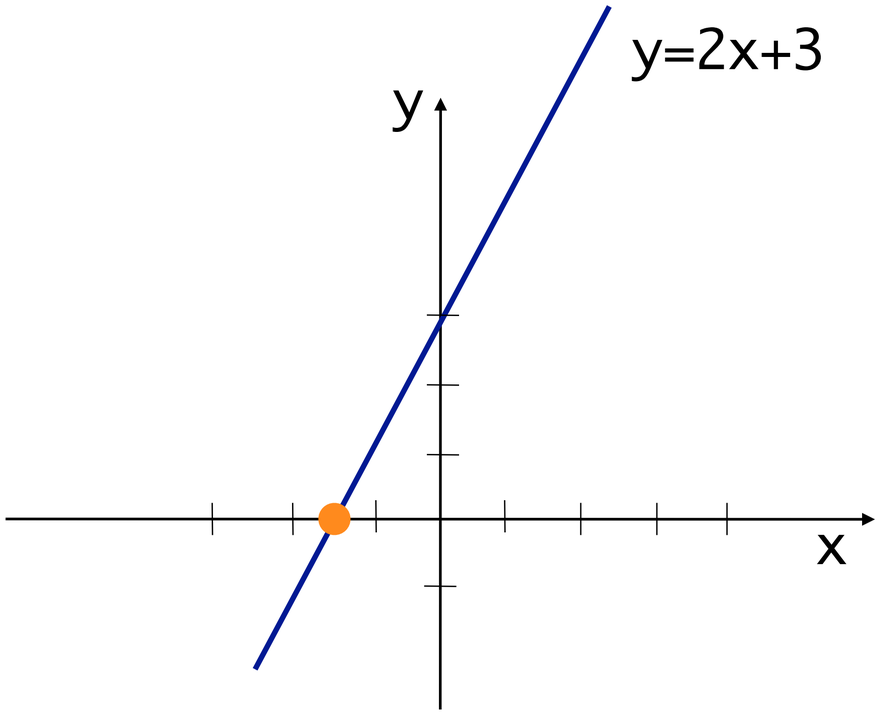
```
(define (search f a b)
  (define (close-enough? x y)
    (< (abs (- x y)) 0.001))
  (let ((mid (average a b)))
    (if (close-enough? a b)
        mid
        (let ((value (f mid)))
          (cond
            ((> value 0) (search f a mid))
            ((< value 0) (search f mid b))
            (else mid))))))

(define (average x y)
  (/ (+ x y ) 2.0))
```

eindconditie

halveer interval naar
eerste of tweede helft

De half interval methode - trace

```
> (define (test x) (+ (* x 2) 3))
> (search test -3 1)
-1.5
> (search test -3 2)
-1.50006103515625
```




```
> (trace search)
> (search test -3 1)
>(search #<procedure:test> -3 1)
>(search #<procedure:test> -3 -1.0)
>(search #<procedure:test> -2.0 -1.0)
<-1.5
-1.5
> (search test -3 2)
>(search #<procedure:test> -3 2)
>(search #<procedure:test> -3 -0.5)
>(search #<procedure:test> -1.75 -0.5)
>(search #<procedure:test> -1.75 -1.125)
>(search #<procedure:test> -1.75 -1.4375)
>(search #<procedure:test> -1.59375 -1.4375)
>(search #<procedure:test> -1.515625 -1.4375)
>(search #<procedure:test> -1.515625 -1.4765625)
>(search #<procedure:test> -1.515625 -1.49609375)
>(search #<procedure:test> -1.505859375 -1.49609375)
>(search #<procedure:test> -1.5009765625 -1.49609375)
>(search #<procedure:test> -1.5009765625 -1.49853515625)
>(search #<procedure:test> -1.5009765625 -1.499755859375)
>(search #<procedure:test> -1.5003662109375 -1.499755859375)
<-1.50006103515625
-1.50006103515625
```

Fixpunt van een functie benaderen

Definition: x is fixpoint of f if x = f(x)

Procedure: for many functions f the fixpoint can be approximated by

...f(f(f(f(f(f(x))))))...

Fixpunt

```
(define (fixpoint f)
  (define (close-enough? x y)
    (< (abs (- x y)) 0.0001))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try 1.0))
```

eindconditie

bereken de volgende
(f guess)

```
> (fixpoint cos)
0.7390547907469174
> (cos 0.7390547907469174)
0.7391055719265363
```

Fixpunt - trace

```
(define (fixpoint f)
  (define (close-enough? x y)
    (< (abs (- x y)) 0.0001))
  (define (try guess)
    (let ((next (f guess)))
  (if (close-enough? guess next)
      next
      (try next))))
  (trace try)
  (try 1.0))
```

```
> (fixpoint cos)
0.739054790746917 4
> (cos 0.739054790746917 4)
0.739105571926536 3
```

```
> (fixpoint cos)
>(try 1.0)
>(try 0.5403023058681398)
>(try 0.8575532158463934)
>(try 0.6542897904977791)
>(try 0.7934803587425656)
>(try 0.7013687736227565)
>(try 0.7639596829006542)
>(try 0.7221024250267079)
>(try 0.7504177617637604)
>(try 0.7314040424225099)
>(try 0.7442373549005569)
>(try 0.7356047404363473)
>(try 0.7414250866101093)
>(try 0.7375068905132427)
>(try 0.7401473355678758)
>(try 0.7383692041223231)
>(try 0.7395672022122561)
>(try 0.7387603198742112)
>(try 0.739303892396906)
>(try 0.7389377567153443)
>(try 0.7391843997714937)
>(try 0.7390182624274122)
>(try 0.7391301765296711)
<0.7390547907469174
0.7390547907469174
```

Sqrt berekenen als een fixpunt (1)

$sqrt(x) = y \Leftrightarrow y^2 = x \Leftrightarrow y = x/y$

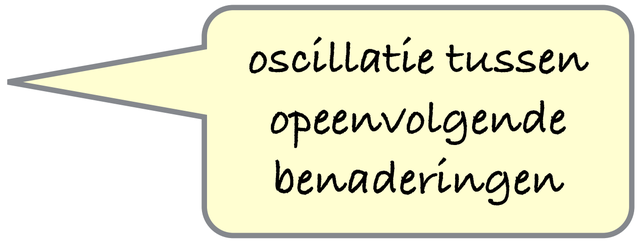
? sqrt(x) can be found as fixpoint of
(lambda (y) (/ x y))

```
(define (sqrt x)
    (fixpoint (lambda (y) (/ x y))))
```

CODE

Sqrt berekenen als een fixpunt(2)

```
> (sqrt 5)
>(try 1.0)
>(try 5.0)
>(try 1.0)
>(try 5.0)
>(try 1.0)
>(try 5.0)
>(try 1.0)
>(try 5.0)
>(try 1.0)
>(try 5.0)
>(try 1.0)
>(try 5.0)
>(try 1.0)
>(try 5.0)
>(try 1.0)
...
>(try 1.0)
>(try 5.0)
⊗⊗ user break
```



!!! does not work try damping,
prevent values to change too much
e.g. use average of x and f(x)

Sqrt berekenen als een fixpunt met demping

standaard demping truuk:

om te vermijden dat functiewaarde in opeenvolgende stappen te bruusk veranderd het gemiddelde met de 'vorige' waarde gebruiken

? sqrt(x) can be found as fixpoint of
(lambda (y) (/ (+ y (/ x y)) 2))

```
(define (sqrt x)
  (fixpoint
    (lambda (y)
      (/ (+ y (/ x y)) 2.0))))
```

```
> (sqrt 5)
>(try 1.0)
>(try 3.0)
>(try 2.3333333333333335)
>(try 2.238095238095238)
>(try 2.236068895643634)
<2.236067977499978
2.236067977499978
```

Les 4: Hogere orde procedures

SESSIE 4

Procedures als terugkeerwaarde: maak vermenigvuldigers

```
> (define (make-multiplier n) (lambda (x) (* x n)))

> (make-multiplier 3)
#<procedure>
> ((make-multiplier 3) 5)
15

> (define double (make-multiplier 2))
> double
#<procedure>
> (define triple (make-multiplier 3))
> triple
#<procedure>
> (double 5)
10
> (triple 5)
15
```

Procedures als terugkeerwaarde: maak sqrt functies

uit les 2

```
(define (make-sqrt p)
  (lambda (x)
    (define (good-enough guess)
      (< (abs (- (square guess) x)) p))
    (define (improve guess)
      (average guess (/ x guess)))
    (define (sqrt-iter guess)
      (if (good-enough guess)
          guess
          (sqrt-iter (improve guess))))
    (sqrt-iter 1.0)))

(define fine-sqrt
  (make-sqrt 0.001))

(define fast-sqrt
  (make-sqrt 0.1))
```

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

```
> (fine-sqrt 4)
2.0000000929222947
> (fast-sqrt 4)
2.000609756097561
```

Demping herbekeken

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

een algemene demping functie

neemt een functie als argument en geeft een andere functie terug als resultaat

```
(define (sqrt x)
  (fixpoint
    (average-damp
      (lambda (y) (/ x y)))))
```

de wortel als het fixpunt van de gedempte functie

```
> (sqrt 4)
>(try 1.0)
>(try 2.5)
>(try 2.05)
>(try 2.000609756097561)
>(try 2.000000929222947)
<2.000000000000002
2.000000000000002
```

Een derde machtswortel

```
(define (cube-root x)
  (fixpoint
    (average-damp
      (lambda (y) (/ x (square y))))))
```

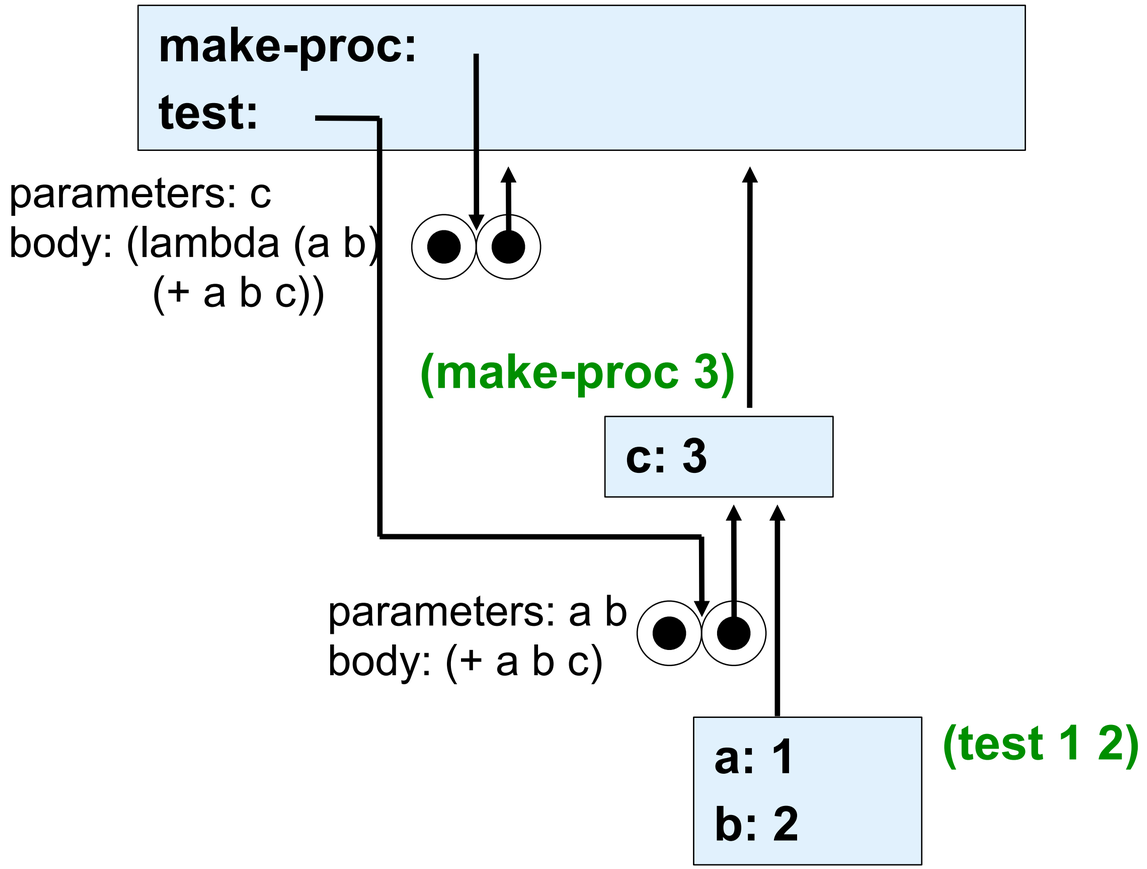
de derdemachtswortel als het fixpunt van de gedempte functie $y \to x/y^2$

```
> (cube-root 8)
>(try 1.0)
>(try 4.5)
>(try 2.447530864197531)
>(try 1.8914996576441667)
>(try 2.063764383263447 6)
>(try 1.9710425766479744)
>(try 2.0151199754332096)
>(try 1.992609760395472)
>(try 2.007362842809587)
>(try 1.998142301706526)
>(try 2.0009314406381735)
>(try 1.999534929963344 7)
>(try 2.0002326972862416)
>(try 1.9998836919616)
>(try 2.000058164165656 3)
<1.999970920454376
1.999970920454376
```

Hogere Orde procedures & lexicale scope

```
(define (make-proc c)
   (lambda (a b) (+ a b c)))

(define test (make-proc 3))
```

```
> test
#<procedure>
> (test 1 2)
6
```



Les 5:
Datastructureringsmechanismen