

Les 9: 'Assignment' en lokale toestand

Sessie 1

Les 9: 'Assignment' en lokale toestand

'Assignment' (toekenning) is in veel traditionele programmeertalen een fundamenteel en noodzakelijk begrip. In Scheme zijn we al ver geraakt zonder maar assignment opent nu een heel nieuw hoofdstuk. We kunnen nu objecten bouwen met lokale toestand.

Overzicht

In deze les wordt eerst en vooral 'assignment' geïntroduceerd.

Dan bouwen we objecten met lokale toestand. Het voorbeeld dat hier stap voor stap wordt uitgewerkt is dat van een bankrekening. Dit voorbeeld introduceert een object geörienteerde stijl van programmeren. Eigenlijk is dit net hetzelfde als wat we in vorige les deden met de complexe getallen. Objecten bevatten lokale data en kunnen boodschappen ontvangen en beantwoorden.

Daarna wordt een toepassing besproken, i.e. het bewaken van functieoproepen door het inpakken van de te bewaken functie.

Assignment

```
> (define x 5)
```

```
> x
```

```
5
```

```
> (+ x 1)
```

```
6
```

```
> x
```

```
5
```

```
> (set! x (+ x 1))
```

```
> x
```

```
6
```

set! special
form geeft
'NIETS' terug

maar heeft
wel een effect

global environment

x: 5

global environment

~~x: 5~~ 6

(set! <name> <new-value>)

Assignment & Call by value

```
(define (increment x)
  (+ x 1))
```

```
(define (increment! x)
  (set! x (+ x 1)))
```

```
> (define a 5)
> a
5
> (increment a)
6
> a
5
```

```
> (define a 5)
> a
5
> (increment! a)
> a
5
```

global environment

a: 5

local environment
(increment! a)

~~x: 5~~ 6

(set! x (+ x 1)))

de assignment gebeurt in de lokale omgeving op de formele parameter

Dingen met toestand

```
(define counter 0)

(define (increment-counter!)
  (set! counter (+ counter 1)))
```

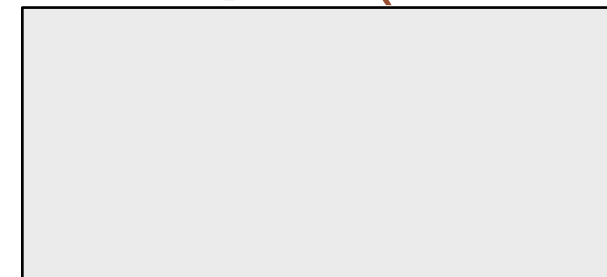
```
> (increment-counter!)
> counter
1
> (increment-counter!)
> (increment-counter!)
> counter
3
```

global environment

counter: ~~0~~ 1

local environment

(increment-counter!)



(set! counter (+ counter 1)))

de assignment gebeurt rechtstreeks
op de globale variabele counter

Functioneel <> Imperatief programmeren

(begin
 <exp-1>
 <exp-2>
 ...
 <exp-n>)

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

> (fac 5)
120

```
(define (fac n)
  (define (fac-iter counter result)
    (if (= counter 0)
        result
        (fac-iter (- counter 1)
                    (* counter result))))
  (fac-iter n 1))
```

> (fac 5)
120

```
(define (fac n)
  (let ((counter n)
        (result 1))
    (define (fac-iter)
      (if (= counter 0)
          result
          (begin
              (set! counter (- counter 1))
              (set! result (* counter result))
              (fac-iter))))
    (fac-iter)))
```

zelf
variabelen
manipuleren

> (fac 5)
0

oé

Functioneel <> Imperatief programmeren

```
(define (fac n)
  (let ((counter n)
        (result 1))
    (define (fac-iter)
      (if (= counter 0)
          result
          (begin
             (set! counter (- counter 1))
             (set! result (* counter result))
             (fac-iter))))
    (fac-iter)))
```

> (fac 5)
0

```
(define (fac n)
  (let ((counter n)
        (result 1))
    (define (fac-iter)
      (if (= counter 0)
          result
          (begin
             (set! result (* counter result))
             (set! counter (- counter 1))
             (fac-iter))))
    (fac-iter)))
```

volgorde van
assignments
is belangrijk

> (fac 5)
120

Geld afhalen van een bankrekening

```
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! balance (- balance amount))
        balance)
      "insufficient funds"))
```

```
(begin
  <exp-1>
  <exp-2>
  ...
  <exp-n>)
```

```
> (withdraw 20)
80
> (withdraw 10)
70
> (withdraw 90)
"insufficient funds"
```

de globale variabele counter stelt het saldo van een bankrekening voor; withdraw is een functie die geld afhaalt van de bankrekening

Geld afhalen met lokale toestand

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin
            (set! balance (- balance amount))
            balance)
          "insufficient funds"))))
```

```
> new-withdraw
#<procedure:new-withdraw>
> (new-withdraw 20)
80
> (new-withdraw 10)
70
> (new-withdraw 90)
"insufficient funds"
```

new-withdraw is een functie die geld afhaalt van een bankrekening; het saldo van de bankrekening is de lokale variabele balance

Processen die geld afhalen creëren

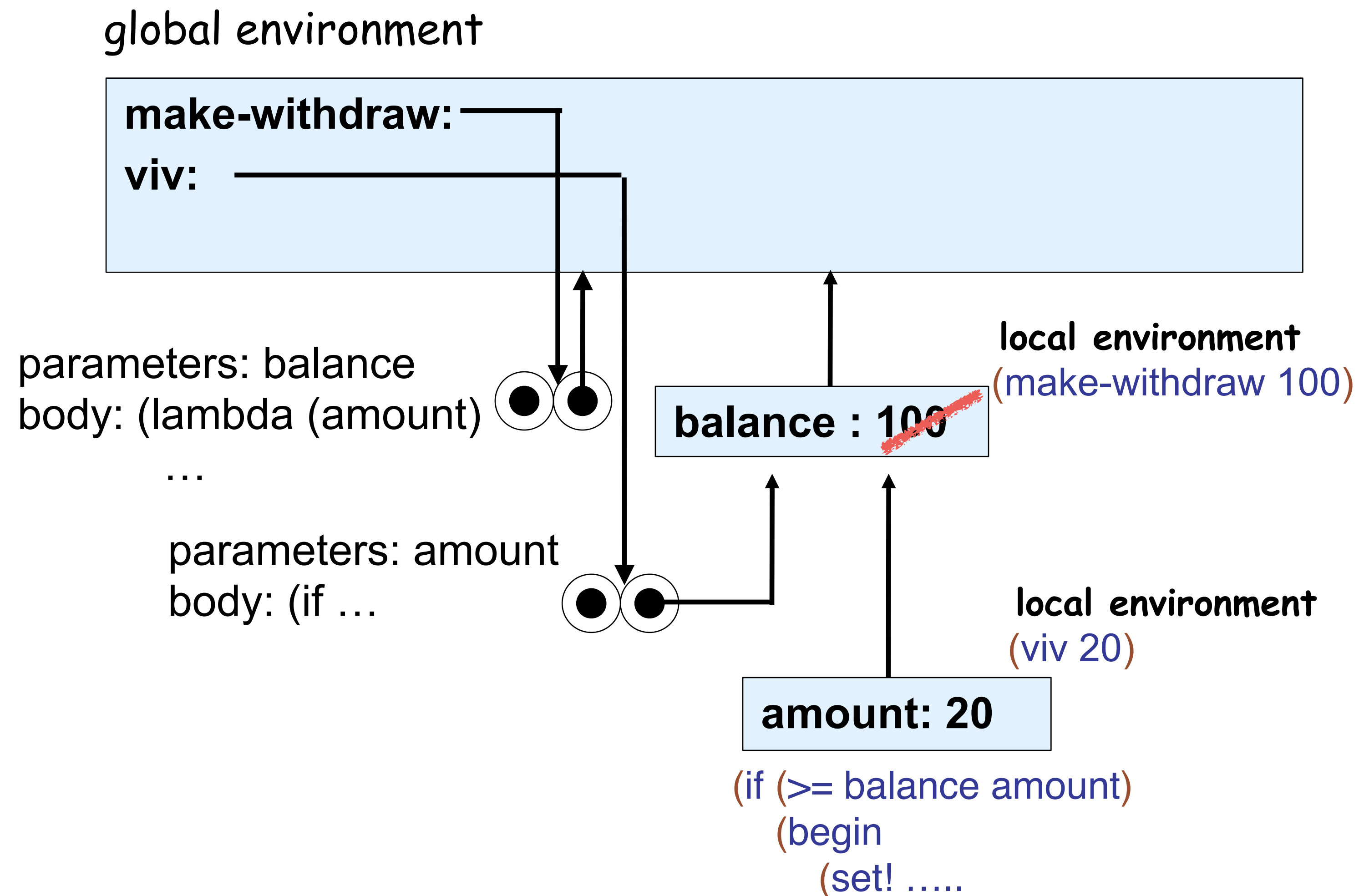
```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "insufficient funds")))
```

make-withdraw is een
hoger orde functie die
withdraw functies
aanmaakt

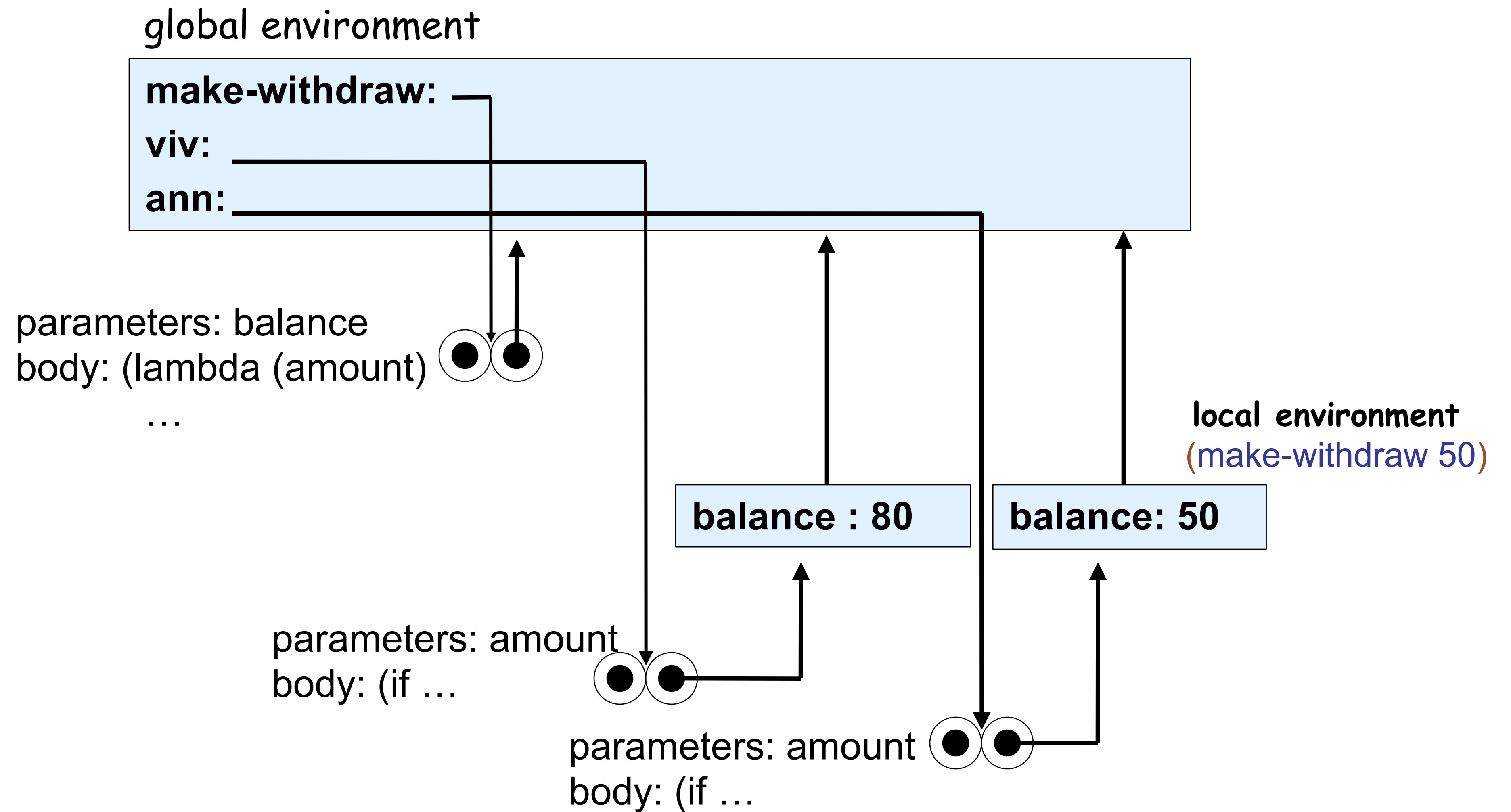
```
> (define viv (make-withdraw 100))
> viv
#<procedure>
> (viv 20)
80
> (viv 10)
70
```

```
> (define ann (make-withdraw 50))
> ann
#<procedure>
> (eq? viv ann)
#f
> (ann 20)
30
> (viv 20)
50
```

Processen die geld afhalen creëren: omgevingsmodel (1)



Processen die geld afhalen creëren omgevingsmodel (2)



Een bankrekening aanmaken

```
(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 'doeiets) 10)
⊗⊗ unknown request
--MAKE-ACCOUNT doiets
```

Message-passing syntactische suiker: twee stijlen

```
(define (send object message par)
  ((object message) par))
```

```
> (define viv (make-account 80))
> (send viv 'withdraw 20)
60
> (send viv 'deposit 50)
110
```

```
(define (send object message)
  ((object (car message)) (cadr message)))
```

```
> (define viv (make-account 80))
> (send viv '(withdraw 20))
60
> (send viv '(deposit 50))
110
```

Les 9: 'Assignment' en lokale toestand

Sessie 2

Een gesofisticeerde bankrekening

```
(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 (status) balance)
  (define (for-dep amount value)
    (set! balance (+ balance (* amount value)))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'status) status)
          ((eq? m 'for-dep) for-dep)
          (else (error "unknown request —MAKE-ACCOUNT" m))))
  dispatch)
```

Met en zonder syntactische suiker

```
> (define viv (make-account 80))  
> ((viv 'deposit) 30)  
110  
> ((viv 'status))  
110  
> ((viv 'for-dep) 20 20)  
510
```

moet boodschappen
met verschillend
aantal argumenten
aankunnen

```
(define (send object message . pars)  
  (apply (object message) pars))
```

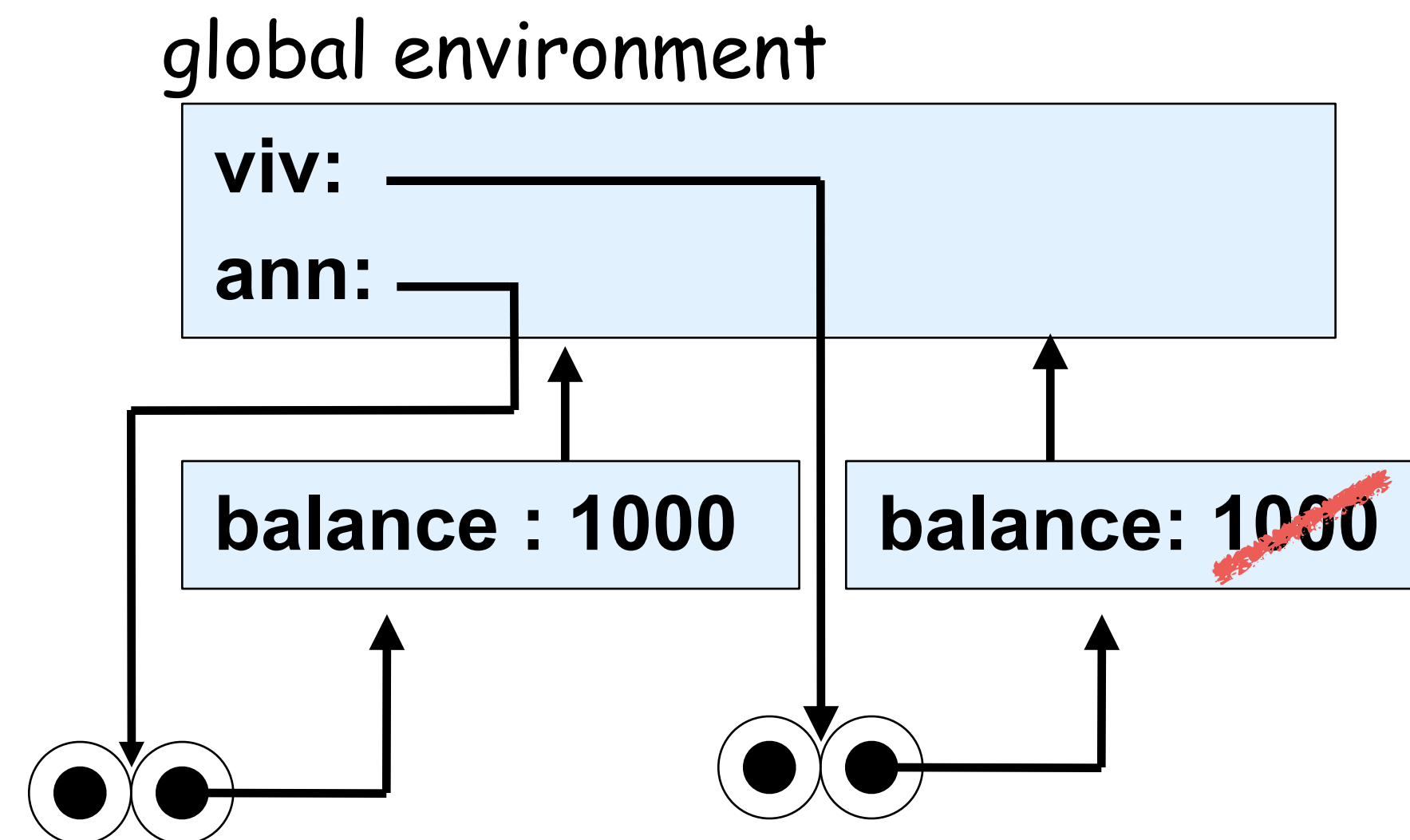
```
> (define viv (make-account 100))  
> (send viv 'withdraw 30)  
70  
> (send viv 'for-dep 20 20)  
470  
> (send viv 'status)  
470
```

```
(define (send object message)  
  (apply (object (car message)) (cdr message)))
```

```
> (define viv (make-account 80))  
> (send viv '(withdraw 30))  
50  
> (send viv '(for-dep 20 20))  
450  
> (send viv '(status))  
450
```

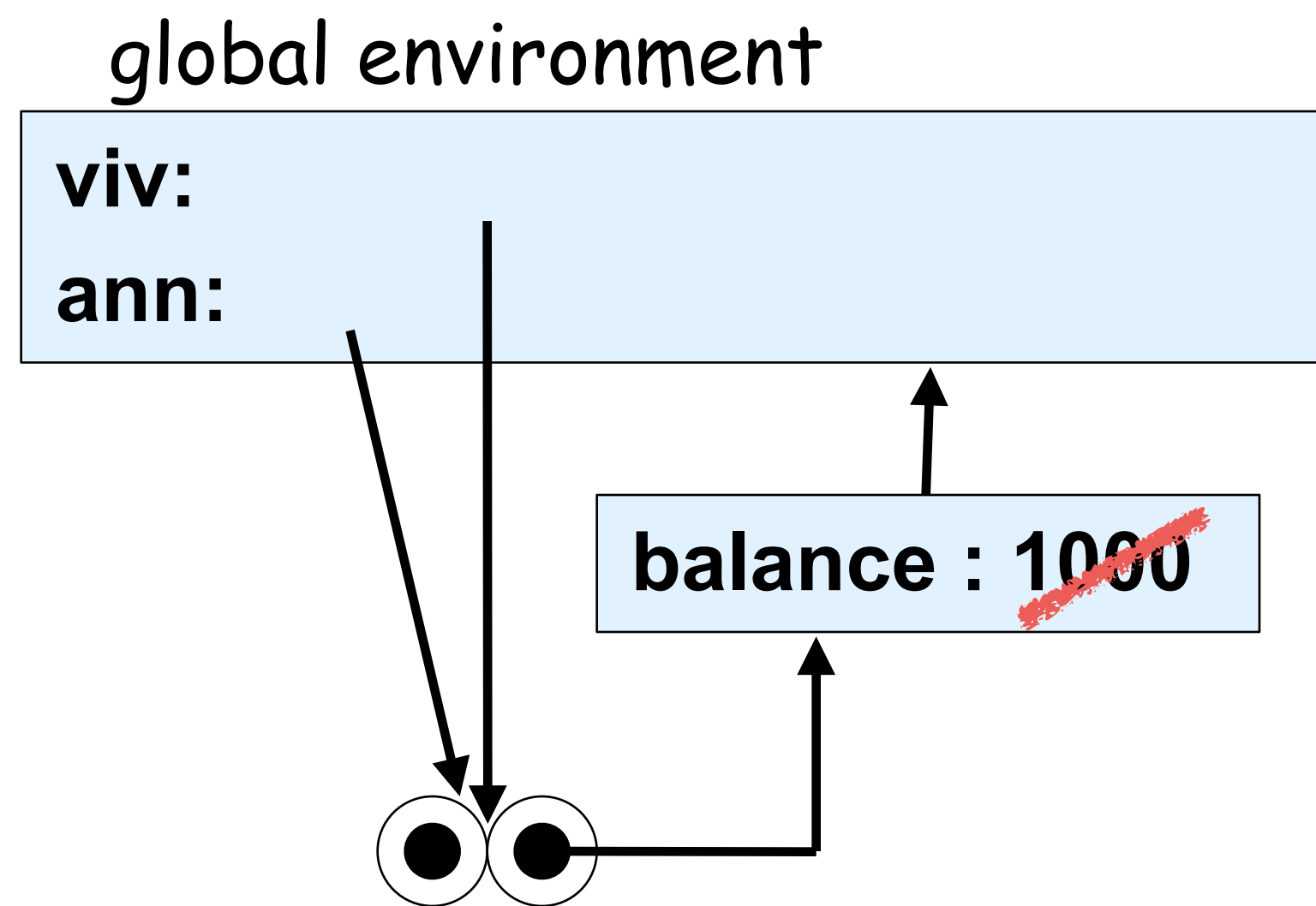
Gelijkheid en verandering: twee bankrekeningen

```
> (define viv (make-account 1000))  
> (define ann (make-account 1000))  
> viv  
#<procedure:dispatch>  
> ann  
#<procedure:dispatch>  
> (eq? viv ann)  
#f  
> ((viv 'deposit) 500)  
1500  
> ((ann 'status))  
1000
```



Gelijkheid en verandering: een gemeenschappelijke bankrekening

```
> (define viv (make-account 1000))  
> (define ann (make-account 1000))  
> (define ann viv)  
> (eq? viv ann)  
#t  
> ((viv 'deposit) 500)  
1500  
> ((ann 'status))  
1500
```



Voorbeeld: monitor functieoproepen

```
(define (p...)
```

```
...
```

```
(f ...))
```

```
(define (f ...)
```

```
(...))
```

```
(define (g ...)
```


```
(f ...)
```

```
(g ...))
```

```
(do ((...))
```

```
(...)
```

```
(f ...))
```



hoe dikwijls wordt
f aangeroepen bij
een run van het
programma?

Voorbeeld: monitor functieoproepen (create monitored function)

```
(define (make-monitored f)
  (let ((count 0))
    (lambda (m)
      (cond
        ((eq? m 'reset) (set! count 0))
        ((eq? m 'how-many-calls) count)
        (else (set! count (+ 1 count))
              (f m)))))))
```

als het argument m
niet één van de
ondersteunde
boodschappen is
wordt de functie
gewoon aangeroepen
op het argument

```
(cond
  (<pred1> <expr11> .. <expr1m>)
  (<pred2> <expr21>) .. <expr2m>)
...
(else <exprn1> .. <exprnm>))
```

Voorbeeld: monitor functieoproepen (try it)

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

mon-double kan
gewoon werken
zoals double

maar kan ook
bijhouden hoeveel
aanroepen er
gebeurd zijn

```
> (define mon-double (make-monitored double))
> double
#<procedure:double>
> mon-double
#<procedure>
> (mon-double 5)
10
> (mon-double 22)
44
> (mon-double 'how-many-calls)
2
> (mon-double 'reset)
0
> (mon-double 'how-many-calls)
0
> (mon-double 1)
2
```

de teller kan ook
op nul gezet
worden voor een
nieuw experiment

Voorbeeld: monitor functieoproepen (re-use function name)

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

```
> (define double (make-monitored double))
> double
#<procedure>
> (double 1)
2
> (double 2)
4
> (double 3)
6
> (double 'how-many-calls)
3
> (double 'reset)
> (double 'how-many-calls)
0
```

de variabele
double wordt
gebonden aan de
gemonitorde
functie

Voorbeeld: monitor functieoproepen (remember and retrieve the original function)

```
(define (make-monitored f)
  (let ((count 0))
    (lambda (m)
      (cond
        ((eq? m 'reset) (set! count 0))
        ((eq? m 'how-many-calls) count)
        ((eq? m 'self) f)
        (else (set! count (+ 1 count))
              (f m))))))
```

de originele functie
is een argument en
dus bereikbaar
binnen de lokale
lambda

op eenvoudig
verzoek wordt ze
naar buiten
gegeven

Voorbeeld: monitor functieoproepen (try again)

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

laat double terug
de naam van de
oorspronkelijke functie
zijn

de oorspronkelijke
functie begrijpt
dit natuurlijk
niet meer

```
> (define double (make-monitored2 double))
> (double 1)
2
> (double 2)
4
> (double 3)
6
> (double 'how-many-calls)
3
> (double 'self)
#<procedure:double>
> (define double (double 'self))
> (double 3)
6
> (double 'how-many-calls)
⊗⊗*: contract violation
expected: number?
given: how-many-calls
argument position: 2nd
```

Les 9: 'Assignment' en lokale toestand

Sessie 2

Voorbeeld: een volgnummersysteem



Bij de lokale bakker is een nieuw systeem ingevoerd dat er voor zorgt dat ook wanneer het druk wordt in de zaak de klanten netjes bediend worden in de volgorde waarin ze binnengekomen zijn.

Bij de ingang kan de klant een nummertje trekken.

Aan de kant van de bakker staat een display en de bakker heeft een toestelletjes met 2 knoppen. Wanneer de bakker klaar is met de bediening van een klant duwt hij op de <next> knop. De display toont dan het nummer van de volgende klant die bediend moet worden.

Als de rol met nummertjes moet vervangen worden duwt de bakker daarna op de <reset> knop. De display toont dan het nummer 0.

Voorbeeld: een volgnummersysteem

Bij de lokale bank is een nieuw systeem ingevoerd dat er voor zorgt dat ook wanneer het druk wordt in de zaak de klanten netjes bediend worden in de volgorde waarin ze binnengekomen zijn.

Bij de ingang staat een toestel met 1 enkele knop.

Wanneer je daarop duwt komt er een tickekje uit met een nummer.

Ook aan de kant van bankbedienden hangt een grote display.

Elke bankbediende heeft een toestel met een <next> knop.

Wanneer een bediende klaar is met de bediening van een klant duwt hij op de <next> knop. Op de display wordt dan naast een loketnummer, het nummer getoond van de volgende klant die moet bediend worden.

De bankmanager heeft een toestel met een <reset> knop.

Na sluitingstijd duwt de bankmanager op een <reset>.

De display toont dan het totaal aantal klanten dat die dag bediend werd en zet ook de tellers van zowel zijn eigen toestel als van het toestel aan de ingang van de winkel op 0.

