IN2040: Funksjonell Programmering

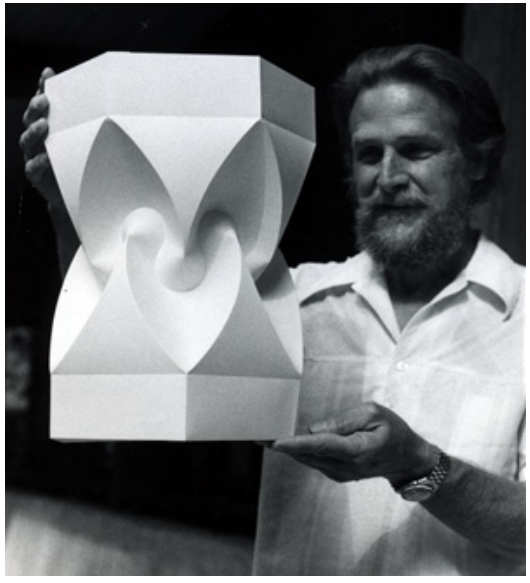
Tilstand og verditilordning

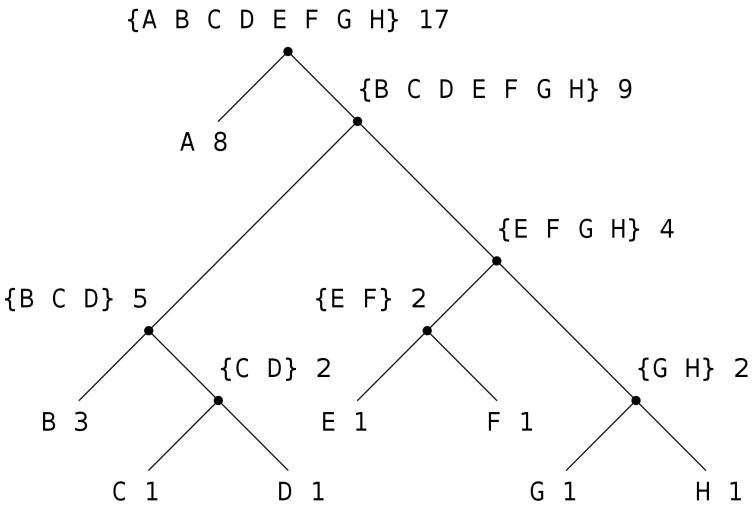
Martin Steffen

Universitetet i Oslo

Uke #6 (27. 09. 2022)

Forrige gang





Repetisjon: Abstraksjonsbarrierer

`en-prosedyre-som-bruker-mengder`

`element-of-set?`, `adjoin-set`, `intersection-set`, `union-set`

`make-tree`, `entry`, `left-branch`, `right-branch`

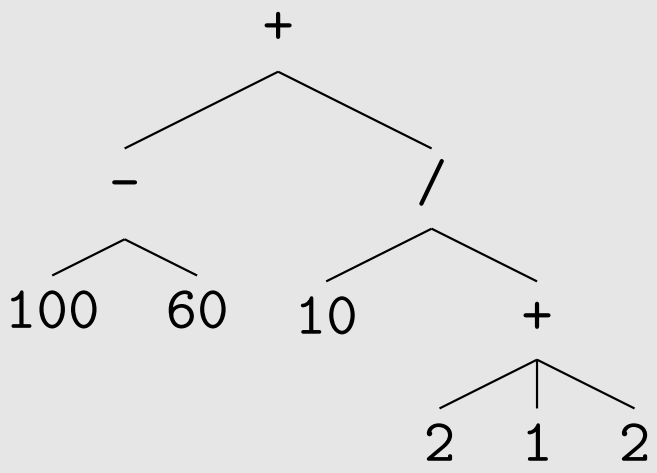
`cons`, `car`, `cdr`, `null?`, `pair?`

`lambda`, eller en annen par-implementasjon?

(En slags) Repetisjon: Kode som trær

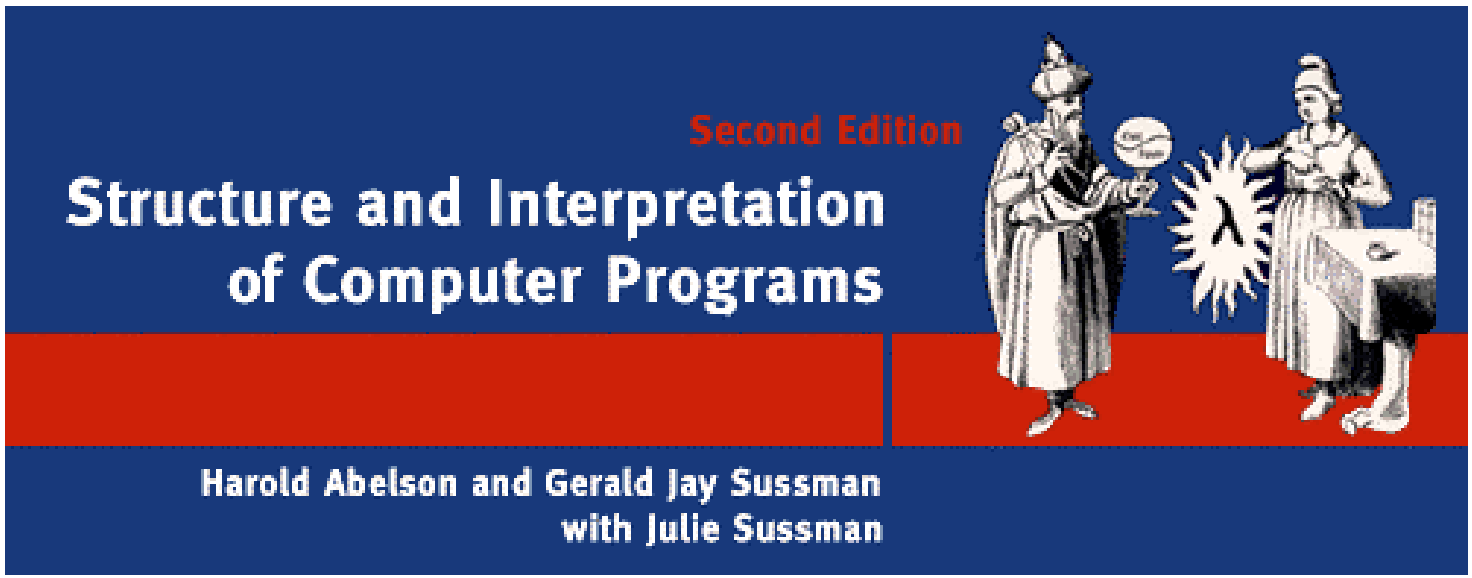
- Ved evaluering oversettes kildekoden i et språk først til et syntakstre.
- Vi har sett at lister kan representere trær i Lisp.
- I Lisp uttrykkes også kode some lister.
- Lisp-kode uttrykker dermed syntaks-trær direkte!

```
(+ (- 100 60)
   (/ 10
      (+ 2 1 2)))
```



I dag

- Vi blar om til kapittel 3 i SICP.
- Tilstand og verditilordning.
- Destruktive operasjoner.
- Prosedyrebasert objektorientering + innkapsling.
- Ny modell for evaluering: omgivelsesmodellen.



Identitet og endring

- I kapittel 3 i SICP tar vi et stort skritt:
- Bryter med ren funksjonell programmering og introduserer elementer fra imperativ programmering:
- Verditilordning som modifiserer objekter.
- Fra et funksjonelt perspektiv: destruktive operasjoner.
  - Når vi endrer tilstanden til et objekt "destruerer" vi det siden det ikke lenger representerer samme verdi.

- Likhet ved identitet vs verdi.
- Til nå har vi kunnet tenke på variabler som *navn på verdier*.
- Fra nå må vi tenke på variabler som *steder man kan lagre verdier*.

Tid og tilstand

- Så langt: et statisk univers.
- Har ikke forholdt oss til tid.
- Sentralt konsept i SICP kap. 3: tilstand.
- Hva mener vi med at noe har tilstand?
- At atferden avhenger av egenskaper som kan forandre seg i forhold til tid og historikk.
- Kap. 3 introduserer også (prosedyrebasert) objektorientering.
- Viktig teknikk for å organisere programmer: Vi kan opprette objekter i programmene våre som tilsvarer entiteter i verden vi ønsker å modellere.

Eksempel på objekt med tilstand: bankkonto

- Klassisk eksempel: bankkonto.
- Saldoen = tilstandsvariabel som oppsummerer historikken (transaksjoner over tid).
- Hva vi kan gjøre, f.eks uttak, avhenger av saldoen.
- Kan definere en variabel `balance` for å representere saldoen.
- Trenger nå en måte å endre verdien symbolet er bundet til: `set!`

```
? (define balance 100)

? (set! balance (- balance 20))

? balance  → 80
```

set!

```
? (define balance 100)

? (set! balance (- balance 20))

? balance  → 80
```

- `set!` er en *special form* som lar oss modifisere variabelbindinger.
- (`set!` $\langle name \rangle$ $\langle value \rangle$)
- Endrer variabelen $\langle name \rangle$ til å få verdien av uttrykket $\langle value \rangle$.
- Brukes for sin *effekt* (tilstandsendringen), ikke *returverdi*.
- Uttales "*sett bæng*".
- Konvensjon i Scheme å bruke `!` ved operasjoner for tilstandsendring med uspesifiert returverdi.

Bankkonto #1

```
(define balance 100)

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

? (withdraw 25) → 75

? (withdraw 25) → 50

? (withdraw 60) → "Insufficient funds"

? (withdraw 15) → 35

? balance → 35

? (set! balance 1000)
```

- Hva er nytt?
- Et kall på `withdraw` har bieffekter utover sin returverdi: Verdien til `balance` endret!
- Samme argumenter, forskjellig returverdi!
- Hvorfor har vi aldri trengt `begin` tidligere?
- Svakhet: Variabelen `balance` er eksponert, kan endres av kode utenfor `withdraw`.

Bankkonto #1 og #2

Først, en ny versjon av #1

```
(define balance 100)

(define withdraw                    ;; cond i stedet for if+begin
  (lambda (amount)
    (cond ((>= balance amount)
           (set! balance (- balance amount))
           balance)
          (else "Insufficient funds"))))
```

Bankkonto #2, med innkapslet saldo

```
(define withdraw
  (let ((balance 100))
    (lambda (amount)
      (cond ((>= balance amount)
             (set! balance (- balance amount))
             balance)
            (else "Insufficient funds")))))
```

Bankkonto #2 (forts.)

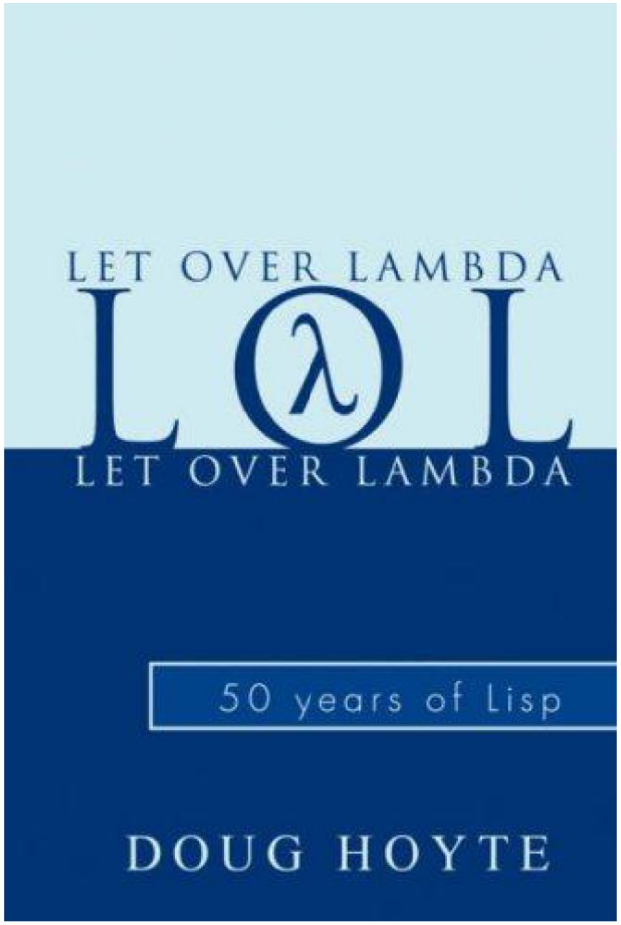
```
(define withdraw
  (let ((balance 100))
    (lambda (amount)
      (cond ((>= balance amount)
             (set! balance (- balance amount))
             balance)
             (else "Insufficient funds")))))

? (withdraw 25) → 75
```

- Gjemmer variabelen med en teknikk som kalles innkapsling i SICP.
  - = closure i Lisp-sjargong.
- `let` etablerer en ny omgivelse med `balance` som lokal variabel.
- `balance` er nå privat: kun tilgang via `withdraw`.
- Merk: Via prosedyren som returneres får vi tilgang til `balance` *utenfor* det som ellers ville være rekkevidden av dens binding (`let`-uttrykket)!
- Kombinasjonen `set!` og lokale variabler lar oss modellere lokal tilstand.

LOL

- Eksempel på et nyttig idiom;
- *let-over-lambda*
- Enkelt eksempel på bruk av innkapsling + verditilordning.

```
(define counter
  (let ((x 0))
    (lambda ()
      (set! x (+ x 1))
      x)))

? (counter) → 1

? (counter) → 2

? (counter) → 3
```



Bankkonto #3

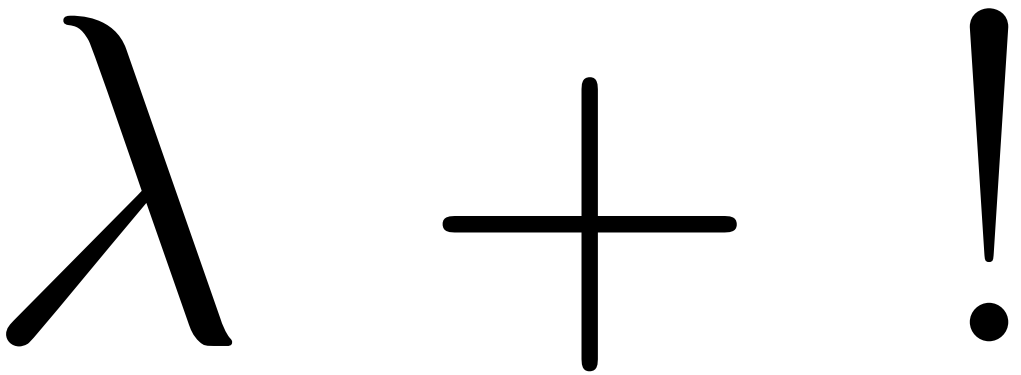
- Svakhet med withdraw: Fortsatt kun én konto.
- make-withdraw: lager nye kontoobjekter.
- Returnerer en anonym prosedyre som innkapsler sin egen saldovariabel:
- Gitt som parameter i stedet for via `let`.
- `w1` og `w2` er distinkte objekter, med distinkte lokale tilstandsvariabler.

```
(define make-withdraw
  (lambda (balance)
    (lambda (amount)
      (cond ((>= balance amount)
             (set! balance (- balance amount))
             balance)
            (else "Insufficient funds")))))

? (make-withdraw 100) → #<procedure>

? (define w1 (make-withdraw 100))

? (define w2 (make-withdraw 200))

? (w1 50) → 50

? (w2 50) → 150
```

Objektorientering via innkapsling og `set!`

- *Closure-oriented programming*:
- I eksemplene våre har vi representerert kontoobjekter som *prosedyrer*.
- Kombinert med *modifisering* av innkapslete lokale variabler for å modellere tilstand.
- Guy Steele om å designe Scheme:

> *One of the conclusions that we reached was that the "object" need not be a primitive notion in a programming language; one can build objects and their behaviour from little more than assignable value cells and good old lambda expressions.*



`set!`

- Hva har skjedd med språket vårt etter at vi innførte `set!`?
- Koden kan ha tilstandsvariabler – lokale eller globale – der bindingen kan endre seg.
- En prosedyre som avhenger av tilstand kan gi ulike returverdier for like argumenter.

- Kan også modifisere tilstandsvariabler: vi sier da at prosedyren har sideeffekter (tidligere trengte vi kun forholde oss til *returverdi*).
- Uttrykk kan evalueres i *sekvenser* der *rekkefølge* har blitt viktig.
- Må tenke på variabler som steder for å lagre verdier (ikke bare navn).

Repetisjon: Substitusjonsmodellen

- Standardregelen for evaluering av uttrykk gjelder også for egendefinerte prosedyrer:
- Først evalueres deluttrykkene, så kalles prosedyren på argumentverdiene.
- I substitusjonsmodellen tenker vi at prosedyreuttrykk omskrives slik:
  - Hele uttrykket erstattes av prosedyre-kroppen fra definisjonen.
  - Parameternavnene erstattes av argumentverdiene.

Eksempel

```
? (define (square x)
    (* x x))

? (square (+ 1 2))

⇒ (square 3)

⇒ (* 3 3)

→ 9
```

Substitusjonsmodellen holder ikke lenger

- La oss forsøke å bruke substitusjonsmodellen her.
- Først et par ting å huske på:
  - Første argumentet til `set!` evalueres ikke (variabelen som skal tilordnes verdi).
  - Ved en *sekvens* av uttrykk er det verdien av det siste som returneres.
- Variablen `balance` brukes tre ganger i kroppen: Substitueres 2. og 3. bruk samtidig blir resultatet feil.

```
? (define (make-withdraw balance)
    (lambda (amount)
      (set! balance (- balance amount))
      balance))

? ((make-withdraw 25) 20)
→ 5
```

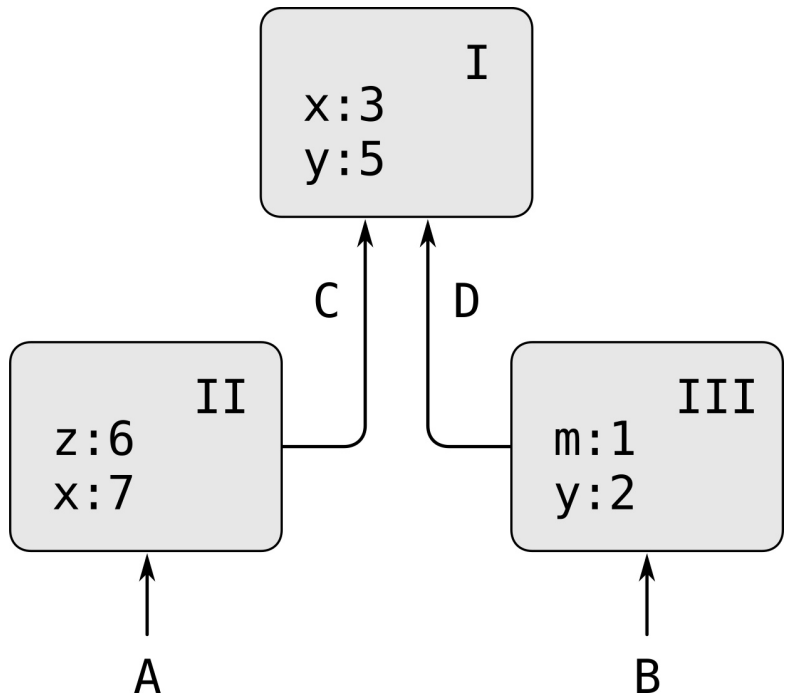
```
? ((make-withdraw 25) 20)

⇒ ((lambda (amount)
     (set! balance (- 25 amount))
     25)
   20)

⇒ (set! balance (- 25 20))
  25

⇒ (set! balance 5)
  25

→ 25
```

En ny modell for evaluering

- Vi skal etablere en ny modell for hvordan kode evalueres.
- Omgivelsesmodellen.
- Sagt litt enkelt er det denne modellen vi skal implementere når vi lager vår egen Scheme-evaluator i kapittel 4.

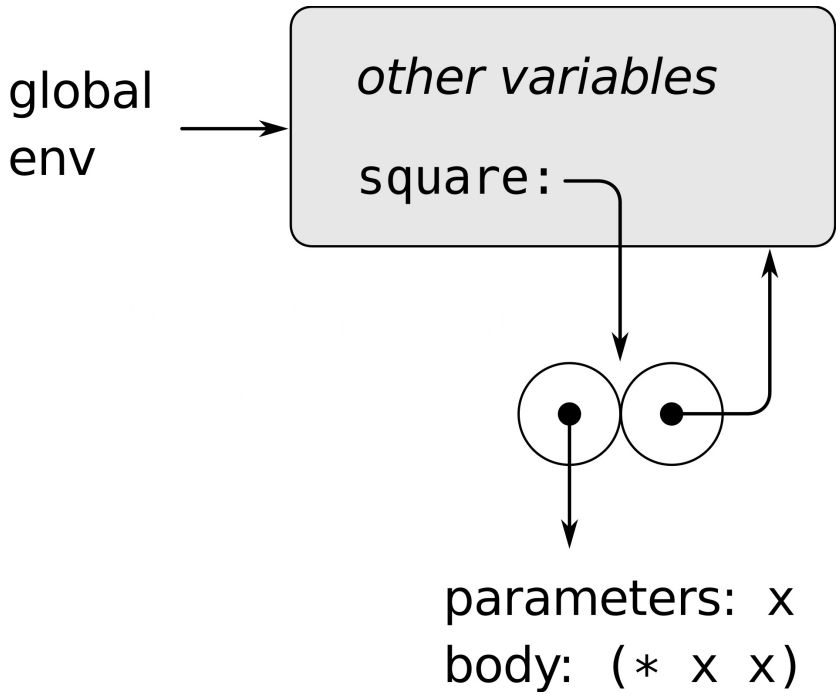
Omgivelser, rammer og bindinger

- En omgivelse er en *sekvens av rammer*.
- En ramme er en tabell som binder variabelnavn til verdier.
- Hver ramme har en peker til sin omsluttende omgivelse.
- Den globale omgivelsen består av én enkelt ramme (med bl.a. bindingene for primitive prosedyrer).
- Variabler har sin verdi relativ til en gitt omgivelse, gitt ved første rammen med en binding.
- Nøstete bindinger for samme navn overskygger hverandre.



Omgivelser og rammer: prosedyrer

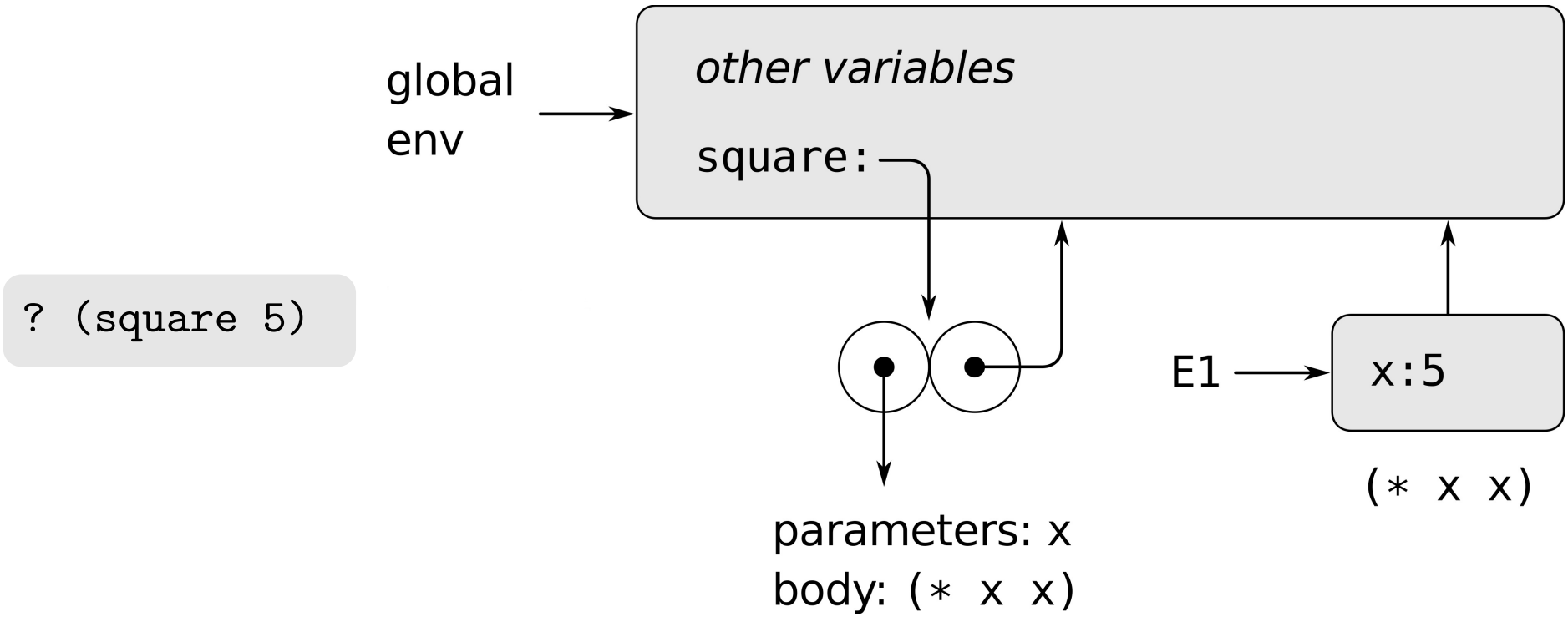
- `define` legger til bindinger i en ramme, `set!` endrer dem.
- En prosedyre lages ved å evaluere et `lambda`-uttrykk i en gitt omgivelse, og representeres ved et "par" som består av:
- Parametere og prosedyrekropp.
- En peker til omgivelsen der den ble opprettet.




```
? (define (square x)
    (* x x))
```

Omgivelser og rammer: prosedyrekall

- Når en prosedyre anvendes, utvides prosedyrens omgivelse med en ny ramme som binder parametrene til argumentverdiene:
- Prosedyrekroppen evalueres så i denne nye omgivelsen.



Omgivelsesmodellen for evaluering

Evalueringsregel for sammensatte uttrykk

- Evaluer deluttrykkene rekursivt.
- Anvend operatorverdien på operandverdiene.

- For å anvende en prosedyre på argumenter,
- utvid omgivelsen til prosedyren med en ny ramme der de formelle parametrene er bundet til argumentene den kalles på,
- og evaluer prosedyrekroppen i denne nye omgivelsen.

- Fortsetter helt til uttrykkene er redusert til atomære uttrykk og primitive prosedyrer (kalles direkte uten ny omgivelse).
- *Special forms* (syntaktiske makroer) har egne evalueringsregler.

Tilbake til innkapsling

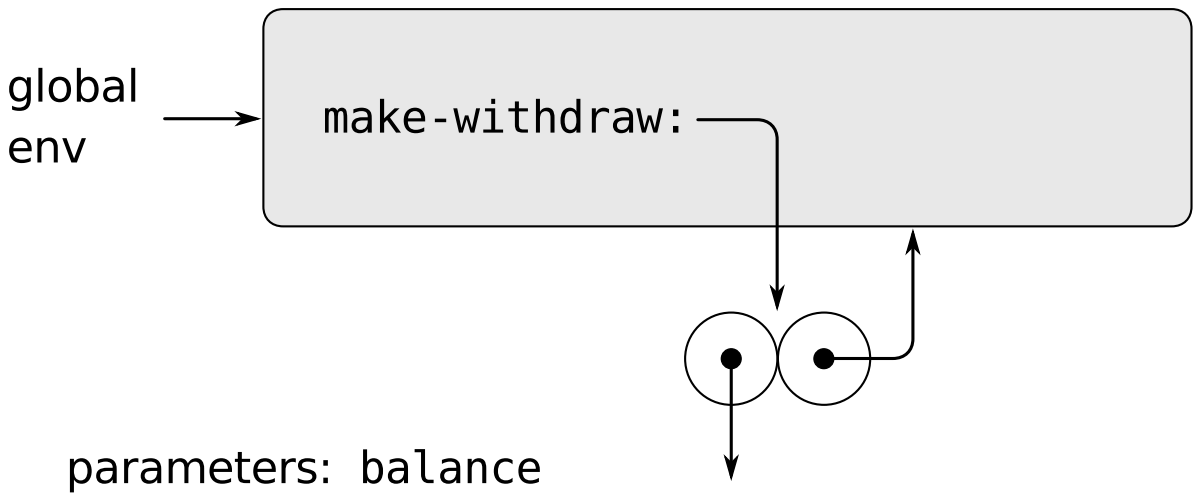
- Innkapsling kan nå avmystifiseres på bakgrunn av omgivelsesmodellen:

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

? (define w1 (make-withdraw 100))

? (w1 50) → 50
```

- Et kall på `make-withdraw` etablerer en ny ramme, som så blir omgivelsen til den anonyme prosedyren den returnerer via `lambda`.

Tilbake til innkapsling (forts.)

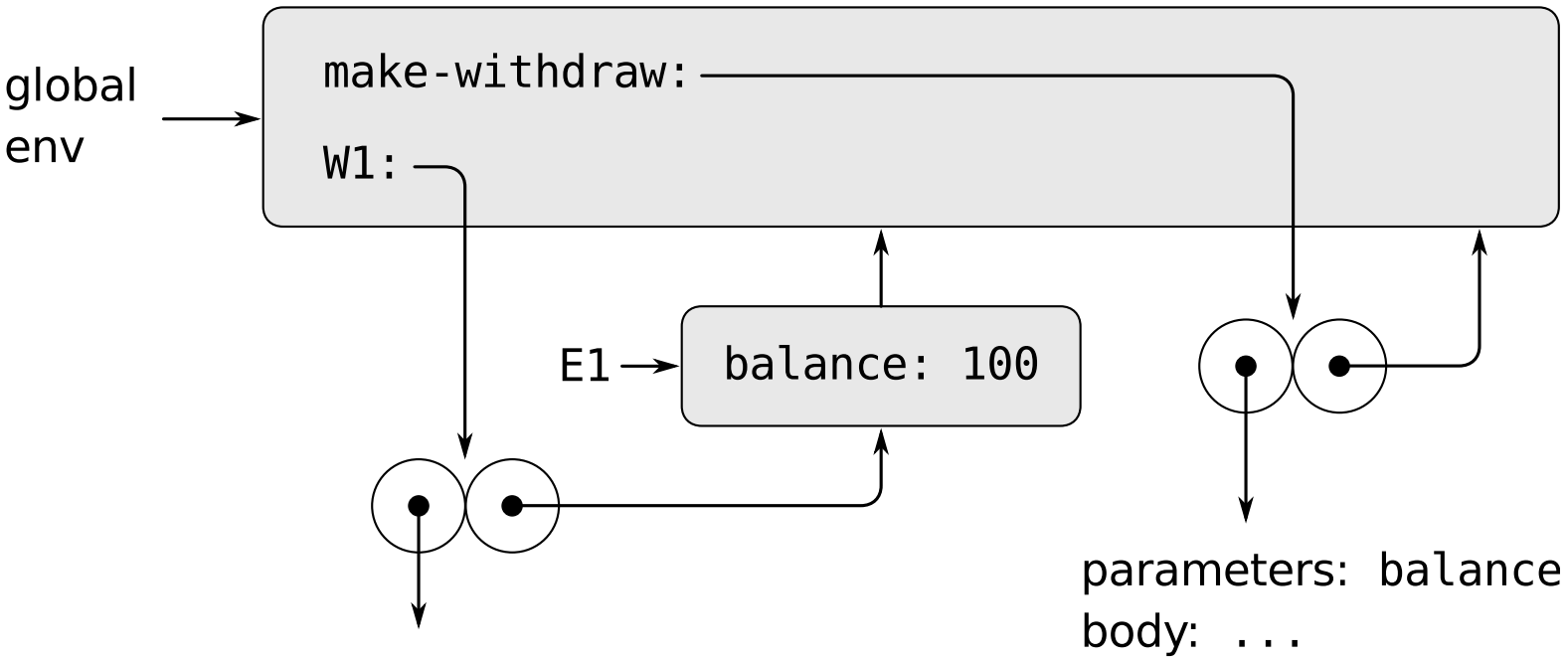
- Resultatet av å definere `make-withdraw` i den globale omgivelsen.




```
parameters: balance
body: (lambda (amount)
        (if (>= balance amount)
            (begin (set! balance (- balance amount))
                   balance)
            "insufficient funds"))
```

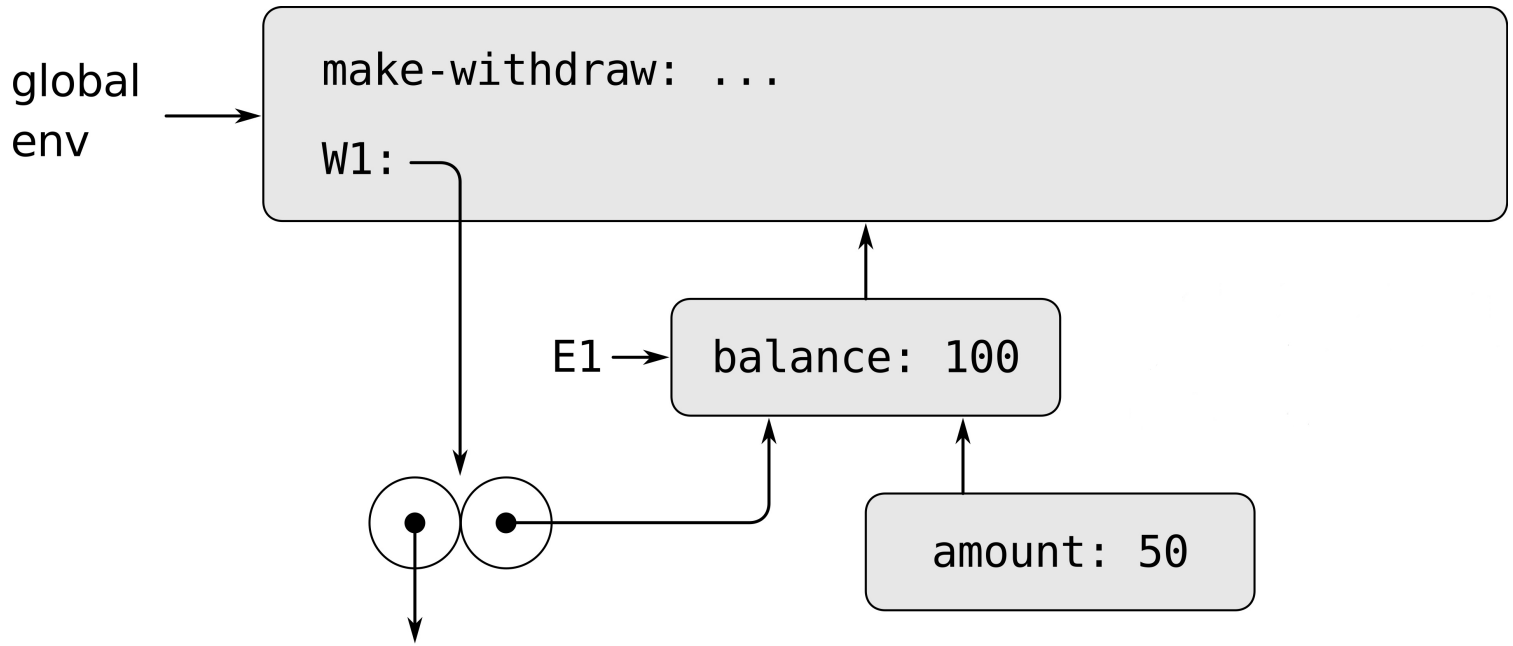
Tilbake til innkapsling (forts.)

```
? (define w1 (make-withdraw 100))
```



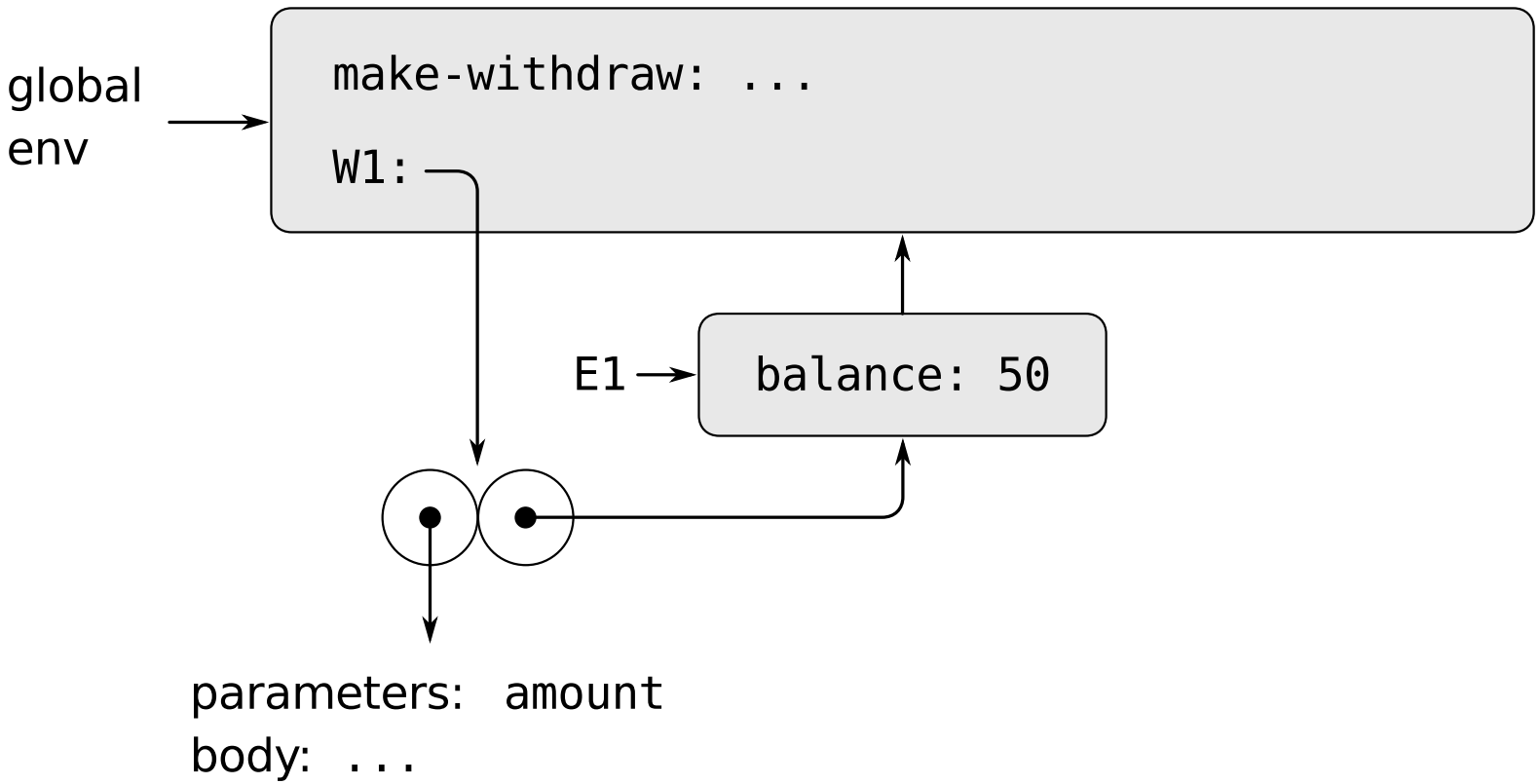
Tilbake til innkapsling (forts.)

```
? (w1 50) → 50
```



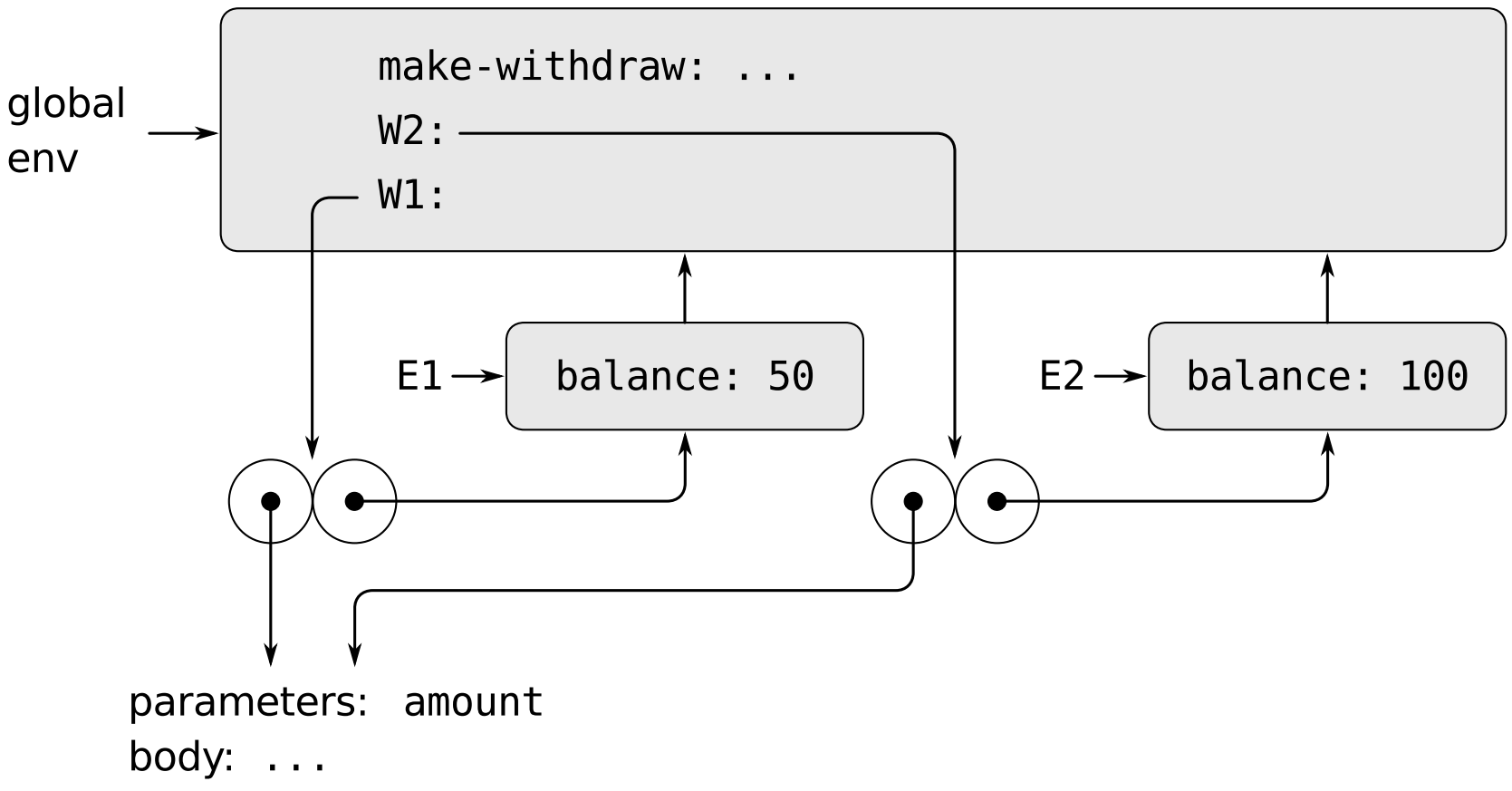
Tilbake til innkapsling (forts.)

- Etter at kallet på `w1` er ferdig utført.



Tilbake til innkapsling (forts.)

```
? (define w2 (make-withdraw 100))
```



Prosedyrebasert objektorientering og *message passing*

```
(define (make-account balance)
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch message)
    (cond ((eq? message 'deposit) deposit)
          ((eq? message 'withdraw) withdraw)
          ((eq? message 'balance) balance)))
  dispatch)

;; Litt syntaktisk sukker:

(define (deposit amount account)
  ((account 'deposit) amount))

(define (withdraw amount account)
  ((account 'withdraw) amount))

(define (balance account)
```

Brukseksempler

```
? (define a1
    (make-account 100))

? (define a2
    (make-account 0))

? ((a1 'deposit) 200)
→ 300

? (a2 'balance) → 0

;; nytt grensesnitt:

? (deposit 200 a1) → 500

? (balance a2) → 0

? (eq? a1 a2) → #f

? (define a3 a2)

? (eq? a3 a2) → #t

? (deposit 42 a3) → 42
```

Likhet (Identitet vs. ekvivalens)

- Typespesifikke predikater; `=` og `string=?`
- `equal?` tester strukturell ekvivalens rekursivt.
- `eq?` gir `#t` hvis argumentene angir samme objekt i minnet.
- Merk at symboler og den tomme lista `'()` er konstanter.

```
? (define bim '(1 2 (3)))

? (define bam '(1 2 (3)))

? (equal? bim bam) → #t

? (eq? bim bam) →  #f

? (define bom bam)    ;; aliasing

? (eq? bom bam) →  #t

? (eq? '(1) '(1)) →   #f

? (eq? '() '()) →  #t

? (eq? 'foo 'foo) →  #t

? (eq? + +) →  #t

? (eq? 2.5 2.5) → ???

? (eq? "foo" "foo") → ???
```