INF2810: Funksjonell Programmering

Eksamensforberedelser

Stephan Oepen & Erik Velldal

Universitetet i Oslo

24. mai 2013

I dag

- Kort oppsummering
- Praktisk om eksamen
- Hvem vant konkurransen om flest oblig-poeng gjennom semesteret?
- Prøveeksamen: Spørsmål og svar.

Noen knagger

- Abstraksjon
- Rekursjon
- Prosedyre-orientering
- Funksjonell programmering

Abstraksjon

- Abstraksjon for å kontrollere kompleksitet (skjuler og isolerer).

Abstraksjon

- Abstraksjon for å kontrollere kompleksitet (skjuler og isolerer).
- Kan se på prosedyre-definisjoner som en abstraksjon:
  - Kode som bruker `square` bryr seg ikke om hvordan den utfører kvadreringen så lenge den returnerer det den lover.

Abstraksjon

- Abstraksjon for å kontrollere kompleksitet (skjuler og isolerer).
- Kan se på prosedyre-definisjoner som en abstraksjon:
  - Kode som bruker `square` bryr seg ikke om hvordan den utfører kvadreringen så lenge den returnerer det den lover.
- Generelle mønstre for bruk av prosedyrer; abstrahert til høyereordens-prosedyrer

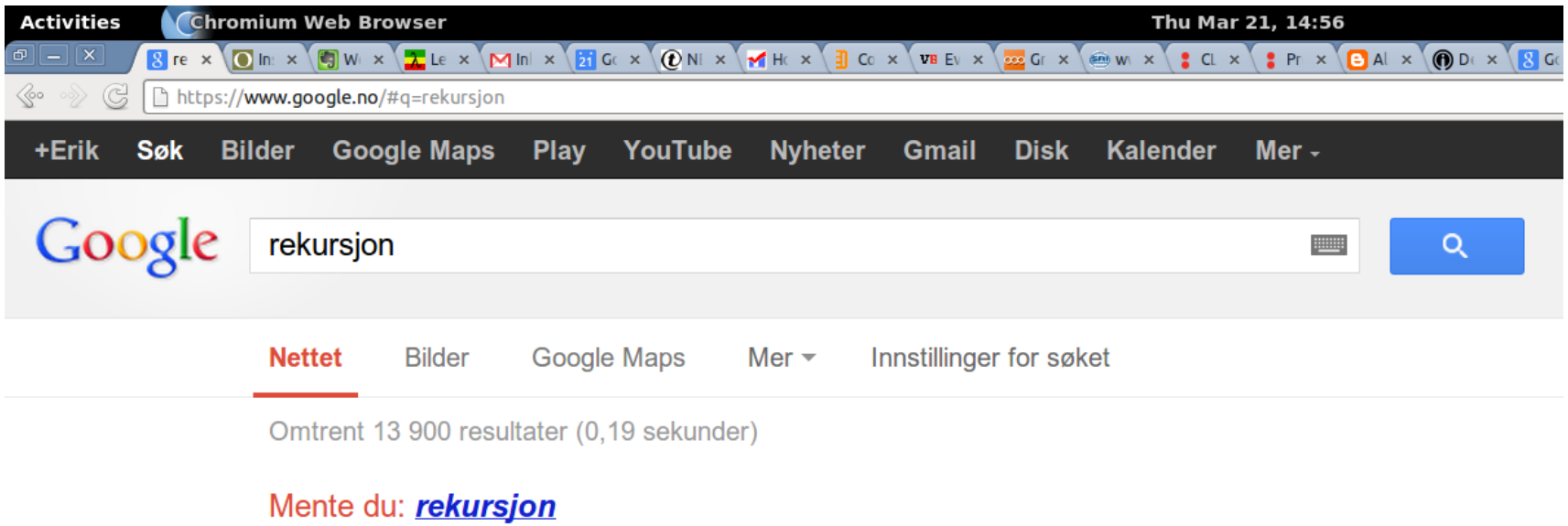
Abstraksjon

- Abstraksjon for å kontrollere kompleksitet (skjuler og isolerer).
- Kan se på prosedyre-definisjoner som en abstraksjon:
  - Kode som bruker `square` bryr seg ikke om hvordan den utfører kvadreringen så lenge den returnerer det den lover.
- Generelle mønstre for bruk av prosedyrer; abstrahert til høyereordens-prosedyrer
- Abstraksjonsbarrierer mellom like deler av et program.
  - Abstrakte datastrukturer definert utfra et sett av operasjoner vi kan utføre på dem;
  - Grensesnittet gjemmer implementasjons-detaljene.
- F.eks; prosedyrer som opererer på en representasjon av mengder; bygget på binærtrær; bygget på lister; bygget på par...

Abstraksjon

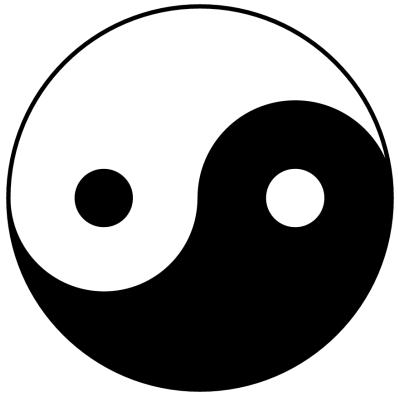
- Abstraksjon for å kontrollere kompleksitet (skjuler og isolerer).
- Kan se på prosedyre-definisjoner som en abstraksjon:
  - Kode som bruker `square` bryr seg ikke om hvordan den utfører kvadreringen så lenge den returnerer det den lover.
- Generelle mønstre for bruk av prosedyrer; abstrahert til høyereordens-prosedyrer
- Abstraksjonsbarrierer mellom like deler av et program.
  - Abstrakte datastrukturer definert utfra et sett av operasjoner vi kan utføre på dem;
  - Grensesnittet gjemmer implementasjons-detaljene.
- F.eks; prosedyrer som opererer på en representasjon av mengder; bygget på binærtrær; bygget på lister; bygget på par...
- Metalingvistisk abstraksjon

Rekursjon


Activities
Chromium Web Browser
Thu Mar 21, 14:56
https://www.google.no/#q=rekursjon
+Erik
Søk
Bilder
Google Maps
Play
YouTube
Nyheter
Gmail
Disk
Kalender
Mer
Google
rekursjon
Nettet
Bilder
Google Maps
Mer
Innstillinger for søket
Omtrent 13 900 resultater (0,19 sekunder)
Mente du: rekursjon

Rekursjon

- Rekursive definisjoner av prosedyrer
  - Direkte rekursjon og gjensidig rekursjon

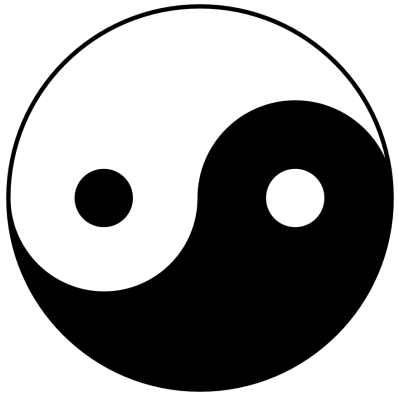


Rekursjon

- Rekursive definisjoner av prosedyrer
  - Direkte rekursjon og gjensidig rekursjon
- Prosedyre vs prosess
  - Rekursive prosesser
  - Iterative prosesser

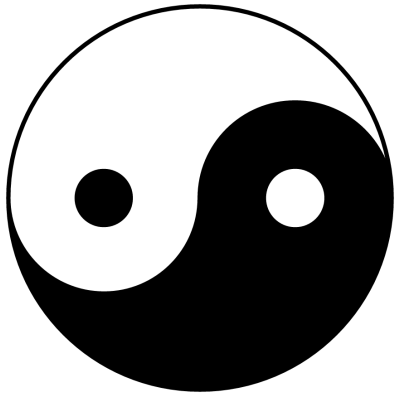
Rekursjon

- Rekursive definisjoner av prosedyrer
  - Direkte rekursjon og gjensidig rekursjon
- Prosedyre vs prosess
  - Rekursive prosesser
  - Iterative prosesser
- Ulike typer idiomer for rekursjon, f.eks;
  - over tall, over lister, over trær, ...
  - oppbygging av ulike typer resultat (sum, produkt, liste, etc.)
  - Halerekursjon, osv



Rekursjon

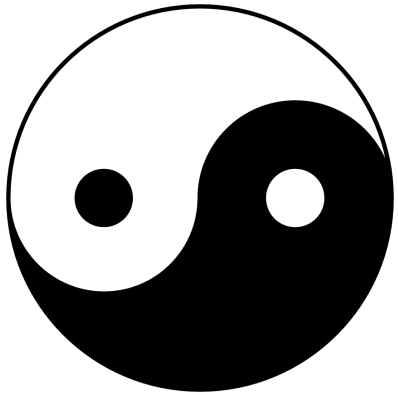
- Rekursive definisjoner av prosedyrer
  - Direkte rekursjon og gjensidig rekursjon
- Prosedyre vs prosess
  - Rekursive prosesser
  - Iterative prosesser



- Ulike typer idiomer for rekursjon, f.eks;
  - over tall, over lister, over trær, ...
  - oppbygging av ulike typer resultat (sum, produkt, liste, etc.)
  - Halerekursjon, osv
- Rekursiv oppbygging av datastrukturer
  - Liste $\equiv$ den tomme lista eller et par der `cdr` er en liste.

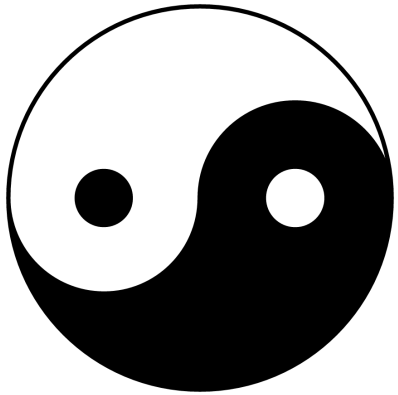
Rekursjon

- Rekursive definisjoner av prosedyrer
  - Direkte rekursjon og gjensidig rekursjon
- Prosedyre vs prosess
  - Rekursive prosesser
  - Iterative prosesser



- Ulike typer idiomer for rekursjon, f.eks;
  - over tall, over lister, over trær, ...
  - oppbygging av ulike typer resultat (sum, produkt, liste, etc.)
  - Halerekursjon, osv
- Rekursiv oppbygging av datastrukturer
  - Liste $\equiv$ den tomme lista eller et par der `cdr` er en liste.
- Rekursivt definerte datastrukturer
  - `(define ones (cons-stream 1 ones))`

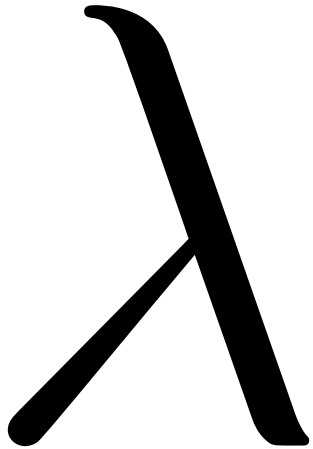
Rekursjon



- Rekursive definisjoner av prosedyrer
  - Direkte rekursjon og gjensidig rekursjon
- Prosedyre vs prosess
  - Rekursive prosesser
  - Iterative prosesser
- Ulike typer idiomer for rekursjon, f.eks;
  - over tall, over lister, over trær, ...
  - oppbygging av ulike typer resultat (sum, produkt, liste, etc.)
  - Halerekursjon, osv
- Rekursiv oppbygging av datastrukturer
  - Liste $\equiv$ den tomme lista eller et par der `cdr` er en liste.
- Rekursivt definerte datastrukturer
  - `(define ones (cons-stream 1 ones))`
- Metasirkulær evaluator
  - Lisp-program for å evaluere Lisp-program

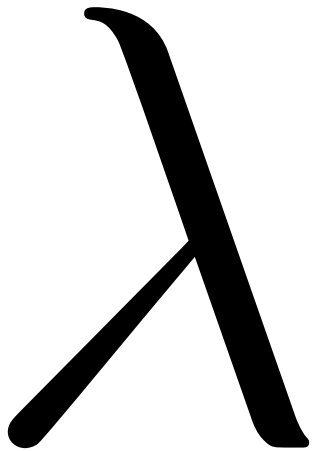
Prosedyre-orientering

- Lambda-uttrykk; navngitte og anonyme prosedyrer



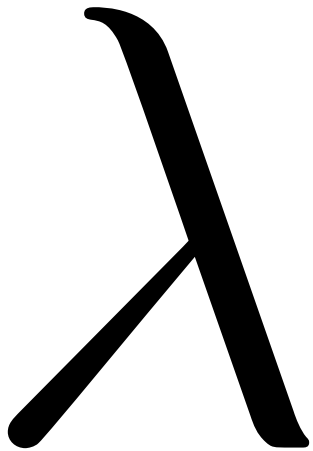
Prosedyre-orientering

- Lambda-uttrykk; navngitte og anonyme prosedyrer
- Prosedyrer som 1.klasses objekter



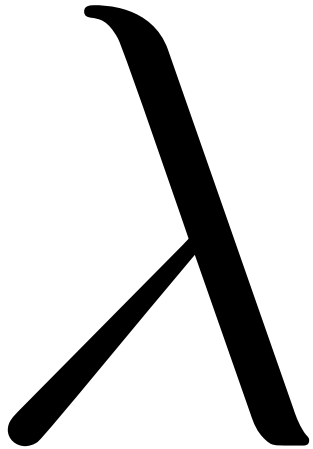
Prosedyre-orientering

- Lambda-uttrykk; navngitte og anonyme prosedyrer
- Prosedyrer som 1.klasses objekter
- Prosedyrer som argumenter og returverdi



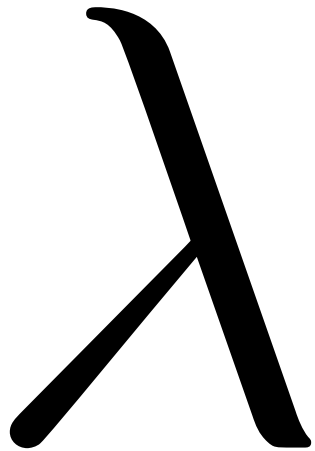
Prosedyre-orientering

- Lambda-uttrykk; navngitte og anonyme prosedyrer
- Prosedyrer som 1.klasses objekter
- Prosedyrer som argumenter og returverdi
- `let` og lokale variabler via anonyme prosedyrer



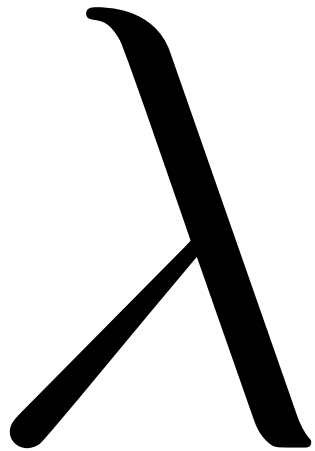
Prosedyre-orientering

- Lambda-uttrykk; navngitte og anonyme prosedyrer
- Prosedyrer som 1.klasses objekter
- Prosedyrer som argumenter og returverdi
- `let` og lokale variabler via anonyme prosedyrer
- Prosedyre-basert objekt-orientering
- $+$ lokal tilstand via innkapsling (f.eks konto-eksemplet)



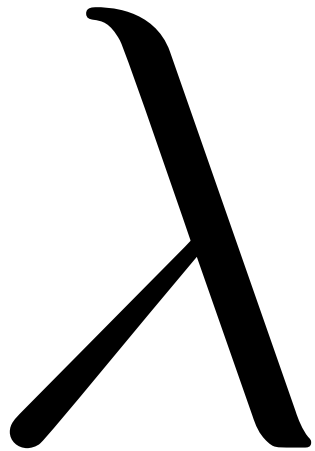
Prosedyre-orientering

- Lambda-uttrykk; navngitte og anonyme prosedyrer
- Prosedyrer som 1.klasses objekter
- Prosedyrer som argumenter og returverdi
- `let` og lokale variabler via anonyme prosedyrer
- Prosedyre-basert objekt-orientering
- $+$ lokal tilstand via innkapsling (f.eks konto-eksemplet)
- Prosedyrer som datastrukturer (f.eks cons-celler)



Prosedyre-orientering

- Lambda-uttrykk; navngitte og anonyme prosedyrer
- Prosedyrer som 1.klasses objekter
- Prosedyrer som argumenter og returverdi
- `let` og lokale variabler via anonyme prosedyrer
- Prosedyre-basert objekt-orientering
- $+$ lokal tilstand via innkapsling (f.eks konto-eksemplet)
- Prosedyrer som datastrukturer (f.eks cons-celler)
- Prosedyrer $=$ data (jf. meta.-evaluatoren)



Funksjonell programmering med noe imperativt attåt

- Hovedfokus: Funksjonell programmering.
- Prosedyrer som kan sees som spesifikasjoner for å beregne matematiske funksjoner:
  - Alltid samme resultat gitt samme argumenter.
  - Beregninger utføres som funksjonelle transformasjoner av data (i stedet for destruktive endringer av tilstandsvariabler).
  - En prosedyre kalles for sin returverdi; uten side-effekter.
  - Semantikken til uttrykk uavhengig av hvor / når de brukes.

Funksjonell programmering med noe imperativt attåt

- Hovedfokus: Funksjonell programmering.
- Prosedyrer som kan sees som spesifikasjoner for å beregne matematiske funksjoner:
  - Alltid samme resultat gitt samme argumenter.
  - Beregninger utføres som funksjonelle transformasjoner av data (i stedet for destruktive endringer av tilstandsvariabler).
  - En prosedyre kalles for sin returverdi; uten side-effekter.
  - Semantikken til uttrykk uavhengig av hvor / når de brukes.
- La så til muterbare datastrukturer og destruktive operasjoner for modellering av endringer i tid og lokal tilstand.

Praktisk om eksamen

- Når: 7. juni kl. 14:30 (4 timer).
- Hvor: Hasle Tennissenter eller Fysikkbygningen (for de som skal bruke PC), sjekk StudentWeb.
- Ingen hjelpemidler.
- Pensum: Forelesningsnotatene + seksjonene vi har gjennomgått fra SICP; se nøyaktig oversikt på emnesiden: www.uio.no/studier/emner/matnat/ifi/INF2810/v13/pensumliste/

Trommevirvel og fanfare!

- Disse sanket inn flest oblig-poeng igjennom semesteret (full pott hele veien):

Trommevirvel og fanfare!

- Disse sanket inn flest oblig-poeng igjennom semesteret (full pott hele veien):

- Johanne Håøy Horn
- Lars Tveito
- Ron Vidar Unhammer
- Erik Winge

Trommevirvel og fanfare!

- Disse sanket inn flest oblig-poeng igjennom semesteret (full pott hele veien):

- Johanne Håøy Horn
- Lars Tveito
- Ron Vidar Unhammer
- Erik Winge
- Premie:

Trommevirvel og fanfare!

- Disse sanket inn flest oblig-poeng igjennom semesteret (full pott hele veien):

- Johanne Håøy Horn
- Lars Tveito
- Ron Vidar Unhammer
- Erik Winge
- Premie:

You might also like...

INF4820 – Algoritmer for kunstig intelligens og naturlige språk

- Generelle teknikker for mønstergjennkjenning, maskinlæring, klassifikasjon, kunnskapsrepresentasjon, dynamisk programmering, memoisering, m.m.
  - Særlig vekt på algoritmer og datastrukturer for analyse av naturlige språk.
- Fokus på praktisk implementasjon, i Common Lisp.
- Gis hver høst av enten Erik eller Stephan, + gjesteforelesere.
  - Høst 2013: Erik + Rebecca Dridan
- www.uio.no/studier/emner/matnat/ifi/INF4820

Prøveeksamen:

oppgaver med løsningsforslag

Oppgave 1

- Fyll inn hva som returneres av uttrykkene under:

```
(define foo 'boff)

(define bar 'baff)

(define (baz foo)
  (let ((bar 'bing))
    (list foo bar)))

(baz bar) →

(baz 'bang) →

(let ((foo 42)
      (bar foo))
  (list bar foo)) →

(let ((foo (lambda (bar) (list bar foo)))
      (bar foo))
  (list (foo bar))) →
```

Oppgave 1

- Fyll inn hva som returneres av uttrykkene under:

```
(define foo 'boff)

(define bar 'baff)

(define (baz foo)
  (let ((bar 'bing))
    (list foo bar)))

(baz bar) → (baff bing)

(baz 'bang) →

(let ((foo 42)
      (bar foo))
  (list bar foo)) →

(let ((foo (lambda (bar) (list bar foo)))
      (bar foo))
  (list (foo bar))) →
```

Oppgave 1

- Fyll inn hva som returneres av uttrykkene under:

```
(define foo 'boff)

(define bar 'baff)

(define (baz foo)
  (let ((bar 'bing))
    (list foo bar)))

(baz bar) → (baff bing)

(baz 'bang) → (bang bing)

(let ((foo 42)
      (bar foo))
  (list bar foo)) →

(let ((foo (lambda (bar) (list bar foo)))
      (bar foo))
  (list (foo bar))) →
```

Oppgave 1

- Fyll inn hva som returneres av uttrykkene under:

```
(define foo 'boff)

(define bar 'baff)

(define (baz foo)
  (let ((bar 'bing))
    (list foo bar)))

(baz bar) → (baff bing)

(baz 'bang) → (bang bing)

(let ((foo 42)
      (bar foo))
  (list bar foo)) → (boff 42)

(let ((foo (lambda (bar) (list bar foo)))
      (bar foo))
  (list (foo bar))) →
```

Oppgave 1

- Fyll inn hva som returneres av uttrykkene under:

```
(define foo 'boff)

(define bar 'baff)

(define (baz foo)
  (let ((bar 'bing))
    (list foo bar)))

(baz bar) → (baff bing)

(baz 'bang) → (bang bing)

(let ((foo 42)
      (bar foo))
  (list bar foo)) → (boff 42)

(let ((foo (lambda (bar) (list bar foo)))
      (bar foo))
  (list (foo bar))) → ((boff boff))
```

Oppgave 2 a+b

- Skriv om følgende let-uttrykk til en ekvivalent form som kun bruker lambda-uttrykk. Oppgi også hva som returneres eller hvilken effekt uttrykket har ved evaluering.

```
(let ((foo (list 1 2))
      (bar (* 3 4)))
  (cons bar foo))
```

```
(let ((foo (list 1 2)))
  (display foo)
  (newline)
  (let ((foo (cons 0 (cdr foo))))
    (display foo)))
```

Oppgave 2 a+b

- Skriv om følgende let-uttrykk til en ekvivalent form som kun bruker lambda-uttrykk. Oppgi også hva som returneres eller hvilken effekt uttrykket har ved evaluering.

```
(let ((foo (list 1 2))
      (bar (* 3 4)))
  (cons bar foo))

((lambda (foo bar)
   (cons bar foo))
 (list 1 2) (* 3 4))
```

→ *(12 1 2)*

```
(let ((foo (list 1 2)))
  (display foo)
  (newline)
  (let ((foo (cons 0 (cdr foo))))
    (display foo)))
```

Oppgave 2 a+b

- Skriv om følgende let-uttrykk til en ekvivalent form som kun bruker lambda-uttrykk. Oppgi også hva som returneres eller hvilken effekt uttrykket har ved evaluering.

```
(let ((foo (list 1 2))
      (bar (* 3 4)))
  (cons bar foo))

((lambda (foo bar)
   (cons bar foo))
 (list 1 2) (* 3 4))

→ (12 1 2)
```

```
(let ((foo (list 1 2)))
  (display foo)
  (newline)
  (let ((foo (cons 0 (cdr foo))))
    (display foo)))

((lambda (foo)
   (display foo)
   (newline)
   ((lambda (foo)
      (display foo))
    (cons 0 (cdr foo))))
 (list 1 2))

⇝ (1 2)
⇝ (0 2)
```

Oppgave 3

- Forklar med én setning hva følgende prosedyre gjør. Vis også et eksempel på kall / returverdi (du må altså selv velge hva argumentene skal være).

```
(define (foo x y z)
  (cond ((null? z) '())
        ((x (car z))
         (cons (y (car z))
               (foo x y (cdr z))))
        (else (foo x y (cdr z)))))
```

Oppgave 3

- Forklar med én setning hva følgende prosedyre gjør. Vis også et eksempel på kall / returverdi (du må altså selv velge hva argumentene skal være).

```
(define (foo x y z)
  (cond ((null? z) '())
        ((x (car z))
         (cons (y (car z))
               (foo x y (cdr z))))
        (else (foo x y (cdr z)))))
```

- Prosedyren returnerer en liste med resultatet av å anvende prosedyren y på hvert element i lista z som tester sant for predikatet x.

Oppgave 3

- Forklar med én setning hva følgende prosedyre gjør. Vis også et eksempel på kall / returverdi (du må altså selv velge hva argumentene skal være).

```
(define (foo x y z)
  (cond ((null? z) '())
        ((x (car z))
         (cons (y (car z))
               (foo x y (cdr z))))
        (else (foo x y (cdr z)))))
```

- Prosedyren returnerer en liste med resultatet av å anvende prosedyren y på hvert element i lista z som tester sant for predikatet x.

```
? (foo odd? (lambda (x) (* x x)) '(1 2 3 4 5 6 7 8 9))
→ (1 9 25 49 81)
```

Oppgave 4, a

- Skriv en prosedyre `compose` som tar to prosedyrer som argument – la oss kalle dem `x` og `y` – og returnerer en ny prosedyre som anvender `x` på resultatet av å anvende `y`. Både `x` og `y`, og den nye prosedyren som returneres, forventer ett argument. Eksempel på bruk:

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

(define (100+ x) (+ x 100))

((compose 1+ 100+) 5) → 106
```

Oppgave 4, a

- Skriv en prosedyre `compose` som tar to prosedyrer som argument – la oss kalle dem `x` og `y` – og returnerer en ny prosedyre som anvender `x` på resultatet av å anvende `y`. Både `x` og `y`, og den nye prosedyren som returneres, forventer ett argument. Eksempel på bruk:

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

(define (100+ x) (+ x 100))

((compose 1+ 100+) 5) → 106
```

```
(define (compose x y)
  (lambda (z) (x (y z))))
```

Oppgave 4, b

- Basert på `compose` skal vi nå skrive en prosedyre `repeat` som tar en prosedyre `x` og et heltall `n` som argumenter, og returnerer en ny prosedyre som anvender prosedyren `x` `n` antall ganger. Eksempel:

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

((repeat 1+ 10) 1) → 11
```

Oppgave 4, b

- Basert på `compose` skal vi nå skrive en prosedyre `repeat` som tar en prosedyre `x` og et heltall `n` som argumenter, og returnerer en ny prosedyre som anvender prosedyren `x` `n` antall ganger. Eksempel:

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

((repeat 1+ 10) 1) → 11
```

```
(define (repeat x n)
  (if (= n 1)
      x
      (compose x (repeat x (- n 1)))))
```

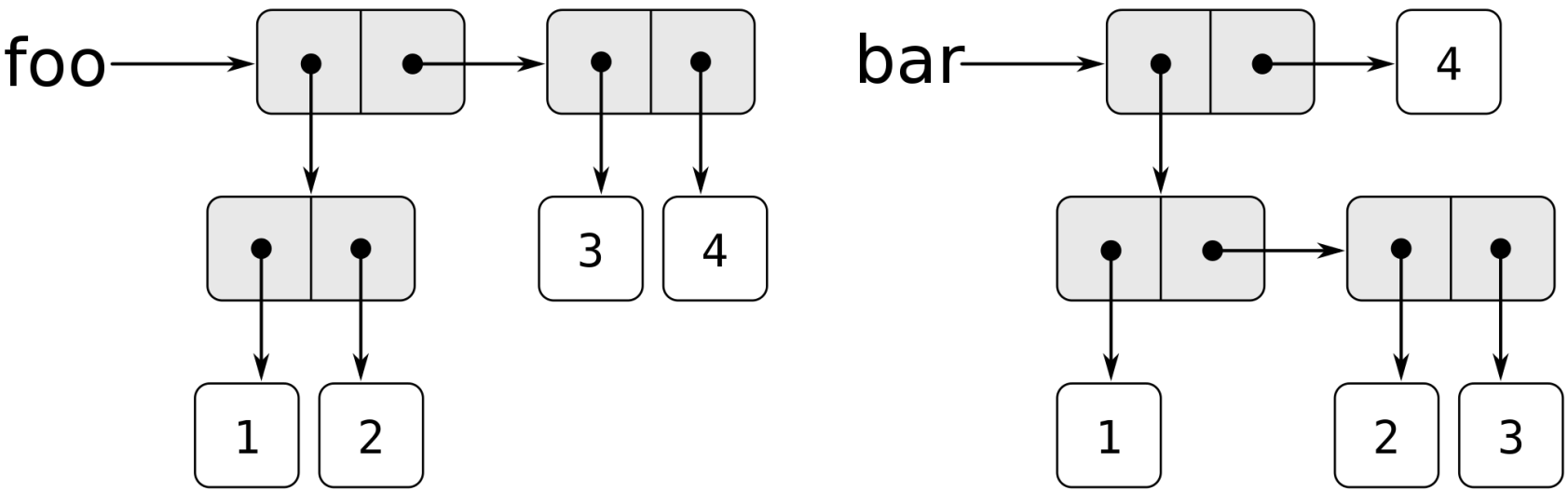
Oppgave 5

- I denne oppgaven skal du skrive to versjoner av en prosedyre `sum-to` som tar et heltall `n` som argument og returnerer summen av heltall fra `1` til `n`. F.eks. vil et kall som `(sum-to 4)` returnere `10`, fordi 1+2+3+4=10. Skriv én versjon som gir opphav til en rekursiv prosess og en annen versjon som gir opphav til en iterativ prosess (den kan gjerne bruke en hjelpeprosedyre).

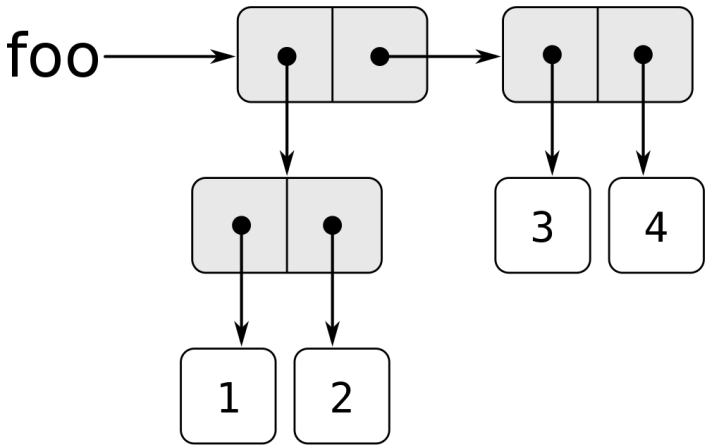
```
(define (sum-to n) ; rekursiv
  (if (zero? n)
      0
      (+ n (sum-to (- n 1)))))

(define (sum-to n) ; iterativ
  (define (iter sum count)
    (if (> count n)
        sum
        (iter (+ sum count) (+ count 1))))
  (iter 0 0))
```

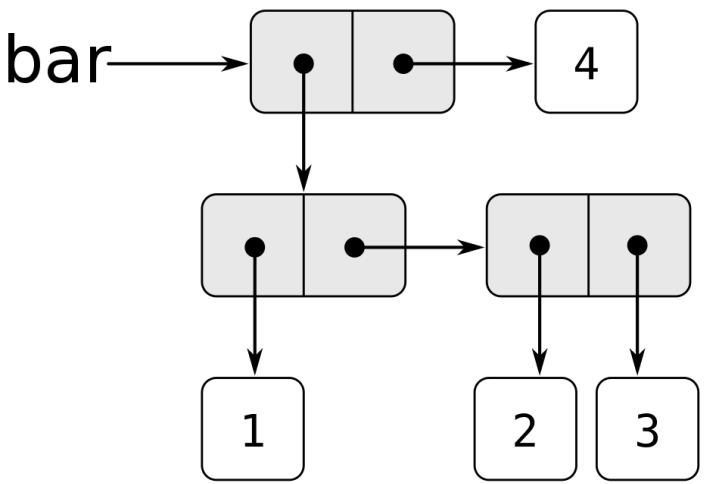
Oppgave 6

La følgende boks-og-peker-diagram beskrive strukturer som er bundet til henholdsvis variabelene `foo` og `bar`. Skriv Scheme-uttrykk som viser hvordan strukturene kan lages med gjentatte kall på `cons`. Skriv også uttrykk som viser hvordan 2 kan erstattes (destruktivt) med 42 i begge.

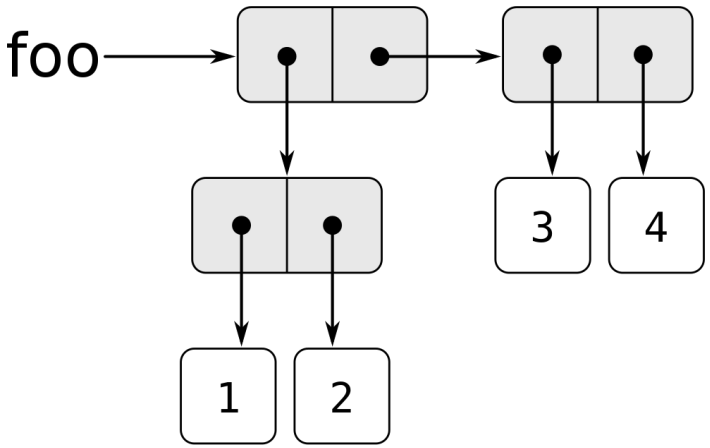


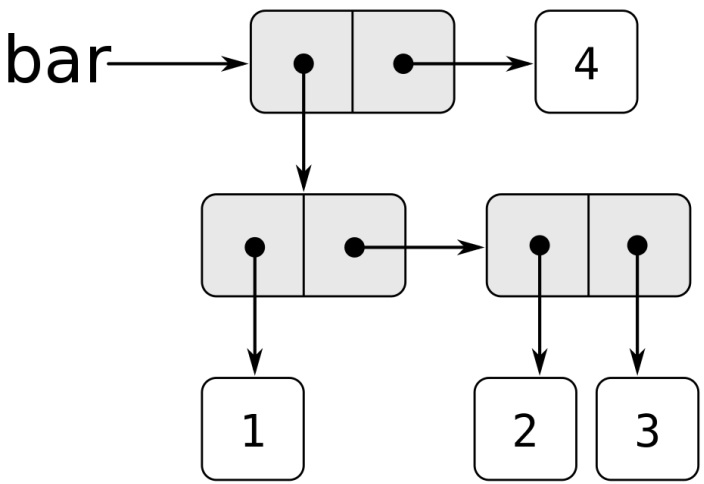
Oppgave 6





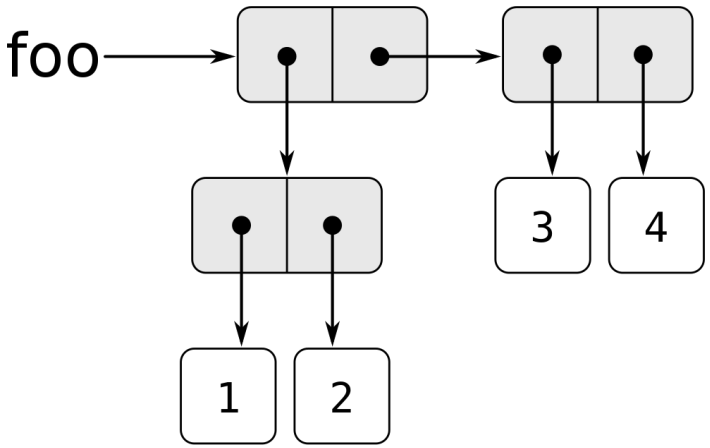
Oppgave 6

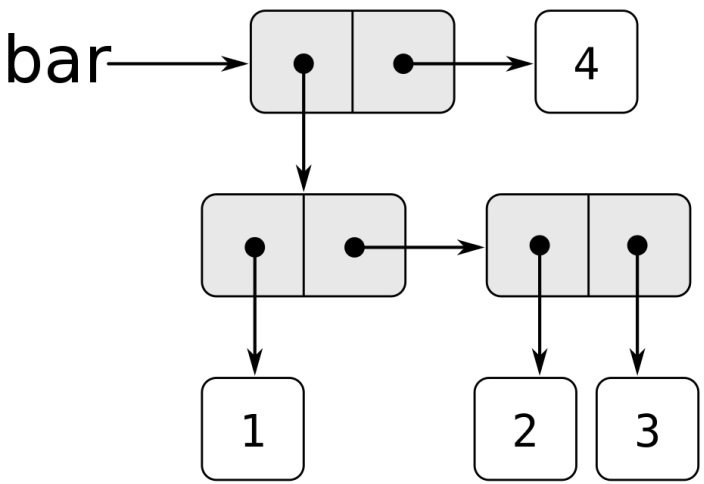





```
? (define foo (cons (cons 1 2)
                    (cons 3 4)))

? foo → ((1 . 2) 3 . 4)
```

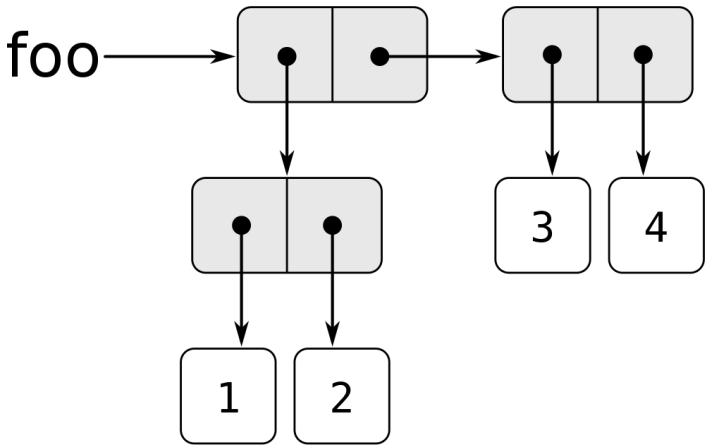
Oppgave 6

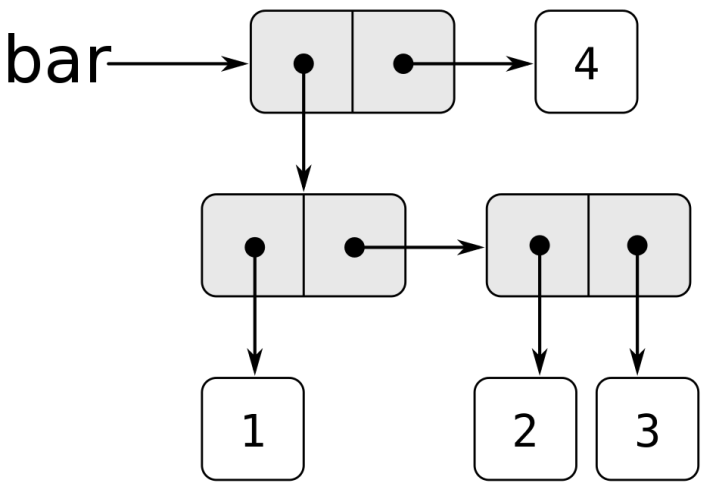





```
? (define foo (cons (cons 1 2)
                    (cons 3 4)))

? foo → ((1 . 2) 3 . 4)

? (set-cdr! (car foo) 42)
```

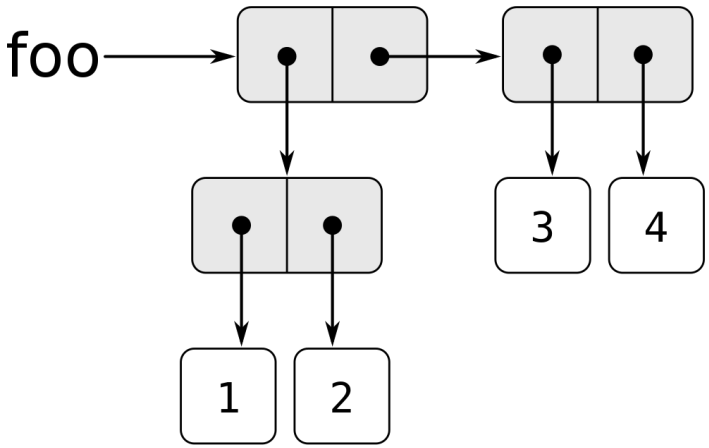
Oppgave 6

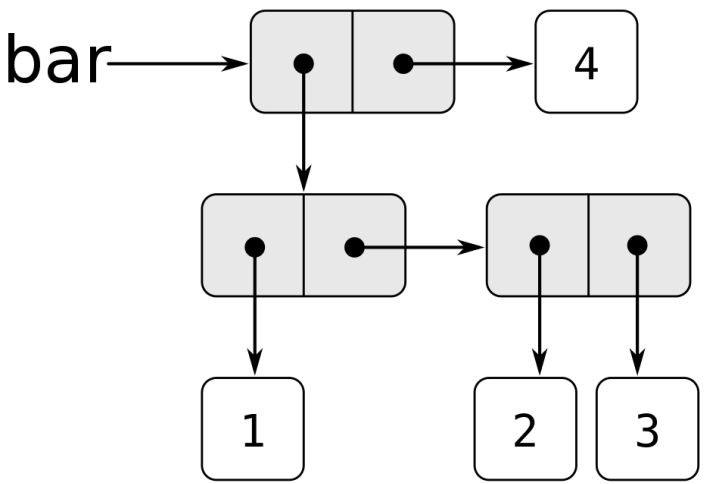





```
? (define foo (cons (cons 1 2)
                    (cons 3 4)))

? foo → ((1 . 2) 3 . 4)

? (set-cdr! (car foo) 42)



? (define bar
    (cons (cons 1
                (cons 2 3))
          4))

? bar → ((1 2 . 3) . 4)
```

Oppgave 6






```
? (define foo (cons (cons 1 2)
                    (cons 3 4)))

? foo → ((1 . 2) 3 . 4)

? (set-cdr! (car foo) 42)



? (define bar
    (cons (cons 1
                (cons 2 3))
          4))

? bar → ((1 2 . 3) . 4)

? (set-car! (cdr (car bar)) 42)
```

Oppgave 7

- Skriv en prosedyre `make-accumulator` som returnerer et nytt prosedyreobjekt som holder rede på en sum (initialisert til 0) og tar som argument en prosedyre som anvendes på sum-variabelen og hvilke andre argumenter som måtte gis sammen med prosedyre-argumentet. Som et spesielt tilfelle skal prosedyren også ta beskjeden `'undo` som lar oss angre et gitt antall utførte kall og gjennopprette summen til hva den var før det (vi må altså også holde rede på historikken her på en eller annen måte). Her er noen eksempler på hvordan akkumulator-objektene skal kunne brukes:

```
? (define acc (make-accumulator))
? (define acc2 (make-accumulator))
? (acc + 5) → 5
? (acc * 2) → 10
? (acc + 20 5 10) → 45
? (acc - 40) → 5
? (acc (lambda (x) (* x x))) → 25
? (acc 'undo 3) → 10
? (acc2 - 10 20) → -30
```

Oppgave 7

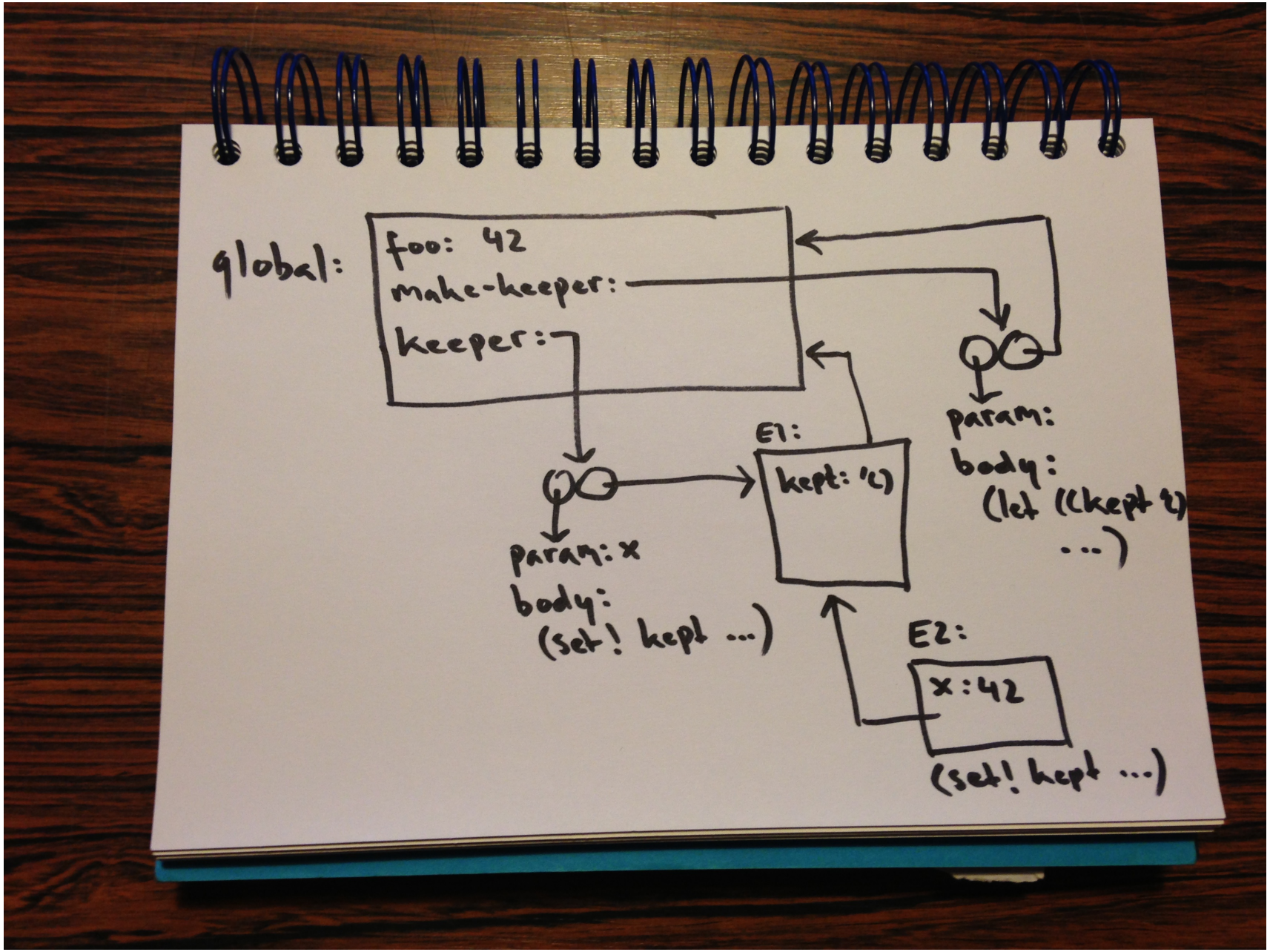
```
(define (make-accumulator)
  (let ((sum 0)
        (history '()))
    (lambda args
      (cond ((procedure? (car args))
             (set! sum (apply (car args) (cons sum (cdr args))))
             (set! history (cons sum history))
             sum)
            ((eq? (car args) 'undo)
             (set! history (nth-cdr history (cadr args)))
             (set! sum (car history))
             sum)))))

(define (nth-cdr list n)
  (if (= n 0)
    list
    (nth-cdr (cdr list) (- n 1))))
```

Oppgave 8

- Her skal vi jobbe med omgivelsesmodellen for evaluering. Tegn et diagram som viser gjeldende omgivelse idet det siste uttrykket i følgende sekvens evalueres:

```
(define foo 42)

(define (make-keeper)
  (let ((kept '()))
    (lambda (x)
      (set! kept (cons x kept))
      kept)))

(define keeper (make-keeper))

(keeper foo)
```

Oppgave 8


global:
foo: 42
make-keeper:
keeper:
E1:
kept: 'c)
param: x
body:
(set! kept ...)
param:
body:
(let ((kept c)
...)
E2:
x: 42
(set! kept ...)

Oppgave 9 (a, b, c)

(a) `(stream-ref integers 4)` $\rightarrow$

Oppgave 9 (a, b, c)

(a) `(stream-ref integers 4)` $\rightarrow$ `5` `;; + kalles fire ganger`

(b) For en hvilken som helst strøm `foo`, beskriv i én setning hva som blir elementene i strømmen som er definert som `(cons-stream 1 (add-streams foo foo))`.

Oppgave 9 (a, b, c)

(a) `(stream-ref integers 4)` $\rightarrow$ `5` `;; + kalles fire ganger`

(b) For en hvilken som helst strøm `foo`, beskriv i én setning hva som blir elementene i strømmen som er definert som `(cons-stream 1 (add-streams foo foo))`.

$\rightarrow$ Resultatstrømmen vil inneholde tallet `1`, so en tallrekke som beregnes som $\texttt{foo}_{i-1} + \texttt{foo}_{i-1}$ for strømposisjon $i > 0$.

(c) Gi en definisjon for fakultetstallene som strøm, slik at `(stream-ref factorials` n`)` $\rightarrow$ $n!$ (dvs. fakultetet av n).

Oppgave 9 (a, b, c)

(a) `(stream-ref integers 4)` → `5` `;; + kalles fire ganger`

(b) For en hvilken som helst strøm `foo`, beskriv i én setning hva som blir elementene i strømmen som er definert som `(cons-stream 1 (add-streams foo foo))`.

→ Resultatstrømmen vil inneholde tallet `1`, so en tallrekke som beregnes som $\texttt{foo}_{i-1} + \texttt{foo}_{i-1}$ for strømposisjon $i > 0$.

(c) Gi en definisjon for fakultetstallene som strøm, slik at `(stream-ref factorials` n`)` → $n!$ (dvs. fakultetet av n).

→
```
(define (stream-mul s1 s2)
  (cons-stream
    (* (stream-car s1) (stream-car s2))
    (stream-mul (stream-cdr s1) (stream-cdr s2))))

(define factorials
  (cons-stream 1 (stream-mul factorials integers)))
```

? `(stream-ref factorials 3)` → `6`

Oppgave 9 (d, e)

(d) Hvor mange elementer inneholder strømmen som defineres som under? Og hvilket resultat vil man få ved testing av denne strømmen med

```
(stream-filter odd? (stream-filter even? integers))
```

Oppgave 9 (d, e)

(d) Hvor mange elementer inneholder strømmen som defineres som under? Og hvilket resultat vil man få ved testing av denne strømmen med `(stream-filter odd? (stream-filter even? integers))`

→ Det er ingen heltall som kan tilfredsstille både `odd?` og `even?`. Derfor er strømmen (konseptuelt) tom. Den ytre `stream-filter` vil derimot aldri returnere en strøm, etter at `integers` er uendelig, og søket 'fremover' i strømmen vil bare fortsette i det uendelige.

(e) Forklar kort relasjonen mellom strømmer og såkalt *lazy evaluation*.

Oppgave 9 (d, e)

(d) Hvor mange elementer inneholder strømmen som defineres som under? Og hvilket resultat vil man få ved testing av denne strømmen med `(stream-filter odd? (stream-filter even? integers))`

→ Det er ingen heltall som kan tilfredsstille både `odd?` og `even?`. Derfor er strømmen (konseptuelt) tom. Den ytre `stream-filter` vil derimot aldri returnere en strøm, etter at `integers` er uendelig, og søket 'fremover' i strømmen vil bare fortsette i det uendelige.

(e) Forklar kort relasjonen mellom strømmer og såkalt *lazy evaluation*.

→ En strøm utsetter evaluering av sin `cdr`, slik at den vil først beregnes når den etterspørs (*on demand*); dette er én måte å realisere *lazy evaluation* på.

Oppgave 10

- Når vi så på hvordan den metasirkulære Scheme-evaluatoren vår kunne endres til å implementere såkalt *normal-order evaluation* eller *lazy evaluation* la vi også til støtte for memoisering av evalueringen av uttrykkene som angir argumentene til prosedyrekall. Forklar kort hvorfor dette ikke var en relevant problemstilling så lenge evaluatoren holdt seg til såkalt *applicative-order evaluation*.

Oppgave 10

- Når vi så på hvordan den metasirkulære Scheme-evaluatoren vår kunne endres til å implementere såkalt *normal-order evaluation* eller *lazy evaluation* la vi også til støtte for memoisering av evalueringen av uttrykkene som angir argumentene til prosedyrekall. Forklar kort hvorfor dette ikke var en relevant problemstilling så lenge evaluatoren holdt seg til såkalt *applicative-order evaluation*.

- Svar: Ved *applicative-order*-strategien evalueres argumentene *før* en prosedyre anvendes, slik at de altså per definisjon kun evalueres én gang. (Ved *normal-order* utsettes evalueringen av uttrykkene som angir argumentene til de trengs i prosedyrekroppen og de kan dermed potensielt ende med å evalueres flere ganger, om vi da ikke memoiserer.)