IN2040: Funksjonell Programmering

Lokale variabler. Og trær.

Martin Steffen

Universitetet i Oslo

4. uke (11. 09. 2023)



Tema forrige uke

- Lister som datastruktur
- `quote`
- Rekursjon på lister
- Høyereordens prosedyrer
  - Prosedyrer som parameter
  - Prosedyrer som returverdi
- `lambda`



Høyereordens sekvensoperasjoner

```
? (define (map proc items)
    (if (null?  items)
        '()
        (cons (proc (car items))
              (map proc (cdr items)))))

? (map square '(-1 -2 0 4 -42))
→ (1 4 0 16 1764)
```

Høyereordens sekvensoperasjoner

```
? (define (map proc items)
    (if (null?  items)
        '()
        (cons (proc (car items))
              (map proc (cdr items)))))

? (map square '(-1 -2 0 4 -42))
→ (1 4 0 16 1764)
```

```
? (define (filter pred items)
    (cond ((null?  items) '())
          ((pred (car items))
           (cons (car items)
                 (filter pred (cdr items))))
          (else (filter pred (cdr items)))))

? (filter (lambda (x)
            (and (> x 2) (< x 8)))
          '(0 1 2 3 4 5 6 7 8 9))
→ (3 4 5 6 7)
```

Høyereordens sekvensoperasjoner

```
? (define (reduce proc init items)
    (if (null?  items)
        init
        (proc (car items)
              (reduce proc init (cdr items)))))

? (reduce + 0 '(1 2 3 4 5)) → 15
```

Høyereordens sekvensoperasjoner

```
? (define (reduce proc init items)
    (if (null?  items)
        init
        (proc (car items)
              (reduce proc init (cdr items)))))

? (reduce + 0 '(1 2 3 4 5)) → 15
```

```
? (define (map proc items)                    ;; map definert med reduce
    (reduce (lambda (item rest)
              (cons (proc item) rest))
            '() items))
```

Høyereordens sekvensoperasjoner

```
? (define (reduce proc init items)
    (if (null?  items)
        init
        (proc (car items)
              (reduce proc init (cdr items)))))

? (reduce + 0 '(1 2 3 4 5)) → 15
```

```
? (define (map proc items)                    ;; map definert med reduce
    (reduce (lambda (item rest)
              (cons (proc item) rest))
            '() items))
```

```
? (define (filter pred items)                 ;; filter definert med reduce
    (reduce (lambda (item rest)
              (if (pred item)
                  (cons item rest)
                  rest))
            '() items))
```

Tema denne uka

- `lambda`, `let` og lokale variabler
- Dataabstraksjon
- Trær og hierarkiske strukturer:
- Lister av lister
- Rekursjon på trær



Lokale variabler og `lambda`

- Nå skal vi se på hvordan vi kan bruke prosedyreapplikasjon med lambda for å få lokale variabler.



Prosedyrer og parametre

```
? (lambda (x)
    (if (even? x) "even!" "odd!"))
→ #<procedure>
```

- `lambda` lager prosedyrer,
- kan være anonyme eller ikke.

Prosedyrer og parametre

```
? (lambda (x)
    (if (even? x) "even!" "odd!"))
→ #<procedure>


? ((lambda (x)
     (if (even? x) "even!" "odd!"))
   7)
→ "odd!"
```

- `lambda` lager prosedyrer,
- kan være anonyme eller ikke.

Prosedyrer og parametre

```
? (lambda (x)
    (if (even? x) "even!" "odd!"))
→ #<procedure>


? ((lambda (x)
     (if (even? x) "even!" "odd!"))
   7)
→ "odd!"
```

- `lambda` lager prosedyrer,
- kan være anonyme eller ikke.
- Parametrene til en prosedyre (som `x` her) er en lokal variabel:
- navngitt variabel der bindingens rekkevidde er prosedyrekroppen.

Prosedyrer og parametre

```
? (lambda (x)
    (if (even? x) "even!" "odd!"))
→ #<procedure>


? ((lambda (x)
     (if (even? x) "even!" "odd!"))
   7)
→ "odd!"
```

- `lambda` lager prosedyrer,
- kan være anonyme eller ikke.
- Parametrene til en prosedyre (som `x` her) er en lokal variabel:
- navngitt variabel der bindingens rekkevidde er prosedyrekroppen.
- Noen ganger ønsker vi å kunne introdusere flere lokale variabler internt i prosedyren: `let`.

Lokale variabler, `let` og `lambda`

Et enkelt eksempel

```
? (let ((a 2)
        (b 3))
    (* a b))
```

→ *6*

Lokale variabler, `let` og `lambda`

Et enkelt eksempel

```
? (let ((a 2)
        (b 3))
    (* a b))
→ 6


;; syntaktisk sukker for:


? ((lambda (a b)
     (* a b))
   2 3)
→ 6
```

- `let`-uttrykk med n variabler.
- Ekvivalent med å kalle et `lambda`-uttrykk med n parametre:
- Variabelbindingene gitt ved prosedyreargumentene.
- Kroppene i uttrykkene ellers like.
- Navnet gjenspeiler matematisk språkbruk ('Let a be the vector $\langle 1, 2, 3 \rangle$.').

Lokale variabler, `let` og `lambda` (forts.)

Et mer realistisk eksempel

```
? (percentages '(10 90 50 50)) → (5 45 25 25)
```

Lokale variabler, `let` og `lambda` (forts.)

Et mer realistisk eksempel

```
? (percentages '(10 90 50 50)) → (5 45 25 25)


(define (percentages items)
  (map (lambda (x) (/ x (/ (reduce + 0 items) 100)))
       items))
```

Lokale variabler, `let` og `lambda` (forts.)



Et mer realistisk eksempel

```
? (percentages '(10 90 50 50)) → (5 45 25 25)


(define (percentages items)
  (map (lambda (x) (/ x (/ (reduce + 0 items) 100)))
       items))


(define (percentages items)
  (let ((sum (/ (reduce + 0 items) 100)))
    (map (lambda (x) (/ x sum))
         items)))
```

Lokale variabler, `let` og `lambda` (forts.)

Et mer realistisk eksempel

```
? (percentages '(10 90 50 50)) → (5 45 25 25)


(define (percentages items)
  (map (lambda (x) (/ x (/ (reduce + 0 items) 100)))
       items))


(define (percentages items)
  (let ((sum (/ (reduce + 0 items) 100)))
    (map (lambda (x) (/ x sum))
         items)))

(define (percentages items)
  ((lambda (sum)
     (map (lambda (x) (/ x sum))
          items))
   (/ (reduce + 0 items) 100)))
```

Lokale variabler, `let` og `lambda` (forts.)

Generell form for `let`

```
(let ((⟨var1⟩ ⟨exp1⟩)
      (⟨var2⟩ ⟨exp2⟩)
       ⋮
      (⟨varn⟩ ⟨expn⟩))
  ⟨body⟩)
```

Ekvivalent `lambda`-uttrykk

```
((lambda (⟨var1⟩ ⟨var2⟩ ... ⟨varn⟩)
   ⟨body⟩)
 ⟨exp1⟩ ⟨exp2⟩ ... ⟨expn⟩)
```

- Rekkevidden til variablene er kroppen til `let`-uttrykket.
- Verdiene ($\langle exp_1 \rangle \ldots \langle exp_n \rangle$) beregnes 'utenfor' `let`-uttrykket:
- Variablene har *ikke* tilgang til hverandre under bindingen.

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)


? (let ((x foo)
        (y foo))
    (list x y))
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)


? (let ((x foo)
        (y foo))
    (list x y))
→ (42 42)
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)


? (let ((x foo)
        (y foo))
    (list x y))
→ (42 42)

? (let ((x foo)
        (y x))
    (list x y))
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)


? (let ((x foo)
        (y foo))
    (list x y))
→ (42 42)

? (let ((x foo)
        (y x))
    (list x y))
→ error: x undefined
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)

? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)

? (let ((x foo)
        (y foo))
    (list x y))
→ (42 42)

? (let ((x foo)
        (y x))
    (list x y))
→ error: x undefined
```

```
? (let ((foo 7)
        (y foo))
    (list foo y))
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)

? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)

? (let ((x foo)
        (y foo))
    (list x y))
→ (42 42)

? (let ((x foo)
        (y x))
    (list x y))
→ error: x undefined
```

```
? (let ((foo 7)
        (y foo))
    (list foo y))
→ (7 42)
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)


? (let ((x foo)
        (y foo))
    (list x y))
→ (42 42)


? (let ((x foo)
        (y x))
    (list x y))
→ error: x undefined
```

```
? (let ((foo 7)
        (y foo))
    (list foo y))
→ (7 42)


? (let ((foo 7))
    (let ((y foo))
      (list foo y)))
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)


? (let ((x foo)
        (y foo))
    (list x y))
→ (42 42)


? (let ((x foo)
        (y x))
    (list x y))
→ error: x undefined
```

```
? (let ((foo 7)
        (y foo))
    (list foo y))
→ (7 42)


? (let ((foo 7))
    (let ((y foo))
      (list foo y)))
→ (7 7)
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)


? (let ((x foo)
        (y foo))
    (list x y))
→ (42 42)

? (let ((x foo)
        (y x))
    (list x y))
→ error: x undefined
```

```
? (let ((foo 7)
        (y foo))
    (list foo y))
→ (7 42)


? (let ((foo 7))
    (let ((y foo))
      (list foo y)))
→ (7 7)


? (let* ((foo 7)
         (y foo))
    (list foo y))
```

Vi øver oss på å forstå bindinger

```
? (define foo 42)


? (let ((x foo)
        (y 1))
    (list x y))
→ (42 1)


? (let ((x foo)
        (y foo))
    (list x y))
→ (42 42)

? (let ((x foo)
        (y x))
    (list x y))
→ error: x undefined
```

```
? (let ((foo 7)
        (y foo))
    (list foo y))
→ (7 42)


? (let ((foo 7))
    (let ((y foo))
      (list foo y)))
→ (7 7)


? (let* ((foo 7)
         (y foo))
    (list foo y))
→ (7 7)
```

Lokale variabler, `lambda` og `let*`

Generell form for `let*`

```
(let* ((⟨var1⟩ ⟨exp1⟩)
       (⟨var2⟩ ⟨exp2⟩)
        ⋮
       (⟨varn⟩ ⟨expn⟩))
  ⟨body⟩)
```

Ekvivalent `lambda`-uttrykk

```
((lambda (⟨var1⟩)
   ((lambda (⟨var2⟩)
      ((lambda (⟨varn⟩)
         ⟨body⟩)
       ⟨expn⟩))
    ⟨exp2⟩))
 ⟨exp1⟩)
```

- Mens `let` binder variablene *parallelt* så gjør `let*` det *sekvensielt*.
- `let*` er en kortform for flere omsluttende `let`- eller `lambda`-uttrykk.



Dataabstraksjon



- Mye av fokuset så langt har vært på *prosedyrer*.
- Modularisering og abstraksjon av *prosesser* (beregninger).
- Like viktig er modularisering og abstraksjon av data ('kunnskap').

Dataabstraksjon



- Mye av fokuset så langt har vært på *prosedyrer*.
- Modularisering og abstraksjon av *prosesser* (beregninger).
- Like viktig er modularisering og abstraksjon av data ('kunnskap').
- Fremover skal vi bruke en del tid på å snakke om data, bl.a:
  - Hvordan definere abstrakte og sammensatte datatyper.
  - Abstraksjonsbarrierer.
  - Hva er egentlig data?
  - Prosedyrer som datastrukturer.
  - Strukturerte data og hierarkiske data (trær)

Noen datatyper

Hvilke typer data kjenner vi i Scheme så langt?

Noen datatyper

Hvilke typer data kjenner vi i Scheme så langt?

	Type	Eksempel
Tall	`integer`, `real`, `rational`	`42`, `3.1415`, `2/3`
Sekvens av tegn	`string`	`"foo bar"`
Symbol	`symbol`	`'foo`, `'hello`
Sannhetsverdi	`boolean`	`#t`, `#f`
Prosedyrer	`procedure`	`+`, `(lambda (x) (* x x))`
Par (2-tuple)	`pair`	`(47 . 11)`
Tom liste	`null`	`'()`

Noen datatyper

Hvilke typer data kjenner vi i Scheme så langt?

	Type	Eksempel
Tall	`integer`, `real`, `rational`	`42`, `3.1415`, `2/3`
Sekvens av tegn	`string`	`"foo bar"`
Symbol	`symbol`	`'foo`, `'hello`
Sannhetsverdi	`boolean`	`#t`, `#f`
Prosedyrer	`procedure`	`+`, `(lambda (x) (* x x))`
Par (2-tuple)	`pair`	`(47 . 11)`
Tom liste	`null`	`'()`

- I tillegg har vi lister: eksempel på en kompleks / sammensatt datatype, bygget på par og den tomme lista.

Noen datatyper

Hvilke typer data kjenner vi i Scheme så langt?

	Type	Eksempel
Tall	`integer`, `real`, `rational`	`42`, `3.1415`, `2/3`
Sekvens av tegn	`string`	`"foo bar"`
Symbol	`symbol`	`'foo`, `'hello`
Sannhetsverdi	`boolean`	`#t`, `#f`
Prosedyrer	`procedure`	`+`, `(lambda (x) (* x x))`
Par (2-tuple)	`pair`	`(47 . 11)`
Tom liste	`null`	`'()`

- I tillegg har vi lister: eksempel på en kompleks / sammensatt datatype, bygget på par og den tomme lista.
- Fremover skal vi jobbe med å definere egne datatyper.
- Viktig skille mellom *abstrakte datatyper* og deres implementasjon.

Hva er data?



- Datatypen manipuleres gjennom et abstrakt grensesnitt som skjuler implentasjonsdetaljene: abstraksjonsbarrieren.
- sentralt i grensesnittet: konstruktorer og selektorer (og predikater)
- Eksempel: Hvordan har vi forholdt oss til datatypen par så langt?
- Kun via grensesnittet den er definert med: `cons`, `car` og `cdr`.
- Denne dataabstraksjonen skjuler en datatypes interne representasjon.
- Abstrakt datatype vs konkret datarepresentasjon:
- Nøyaktig hvordan par er representert internt er ikke viktig, så lenge grensesnittet oppfyller 'kontrakten' sin.
- Vi skal implementere en egen alternativ versjon av datatypen par.

Par som prosedyre!

```
(define (cons x y)
  (lambda (message)
    (cond ((= message 0) x)
          ((= message 1) y))))

(define (car proc)
  (proc 0))

(define (cdr proc)
  (proc 1))

? (cons 1 2)
→ #<procedure>
```

- Selv datastrukturer kan implementeres som prosedyrer!
- Par som prosedyre: Returnerer enten `car`- eller `cdr`-verdien, avhengig av hvilken 'kode' den får (0/1).
- Eksempel på såkalt *message passing*.
- `car` og `cdr` kaller par-prosedyren (laget med `cons`) med 'beskjed' om hvilken verdi som ønskes.

Par som prosedyre!

```
(define (cons x y)
  (lambda (message)
    (cond ((= message 0) x)
          ((= message 1) y))))

(define (car proc)
  (proc 0))

(define (cdr proc)
  (proc 1))

? (cons 1 2)
→ #<procedure>

? (cdr (cons 1 2))
→ 2
```

- Selv datastrukturer kan implementeres som prosedyrer!
- Par som prosedyre: Returnerer enten `car`- eller `cdr`-verdien, avhengig av hvilken 'kode' den får (0/1).
- Eksempel på såkalt *message passing*.
- `car` og `cdr` kaller par-prosedyren (laget med `cons`) med 'beskjed' om hvilken verdi som ønskes.

Par som prosedyre!

```
(define (cons x y)
  (lambda (message)
    (cond ((= message 0) x)
          ((= message 1) y))))

(define (car proc)
  (proc 0))

(define (cdr proc)
  (proc 1))

? (cons 1 2)
→ #<procedure>

? (cdr (cons 1 2))
→ 2
```

- Selv datastrukturer kan implementeres som prosedyrer!
- Par som prosedyre: Returnerer enten `car`- eller `cdr`-verdien, avhengig av hvilken 'kode' den får (0/1).
- Eksempel på såkalt *message passing*.
- `car` og `cdr` kaller par-prosedyren (laget med `cons`) med 'beskjed' om hvilken verdi som ønskes.

Par som prosedyre!

```
(define (cons x y)
  (lambda (message)
    (cond ((= message 0) x)
          ((= message 1) y))))

(define (car proc)
  (proc 0))

(define (cdr proc)
  (proc 1))

? (cons 1 2)
→ #<procedure>

? (cdr (cons 1 2))
→ 2

? (car (cons 1 (cons 2 3)))
→ 1
```

- Selv datastrukturer kan implementeres som prosedyrer!
- Par som prosedyre: Returnerer enten `car`- eller `cdr`-verdien, avhengig av hvilken 'kode' den får (0/1).
- Eksempel på såkalt *message passing*.
- `car` og `cdr` kaller par-prosedyren (laget med `cons`) med 'beskjed' om hvilken verdi som ønskes.

Par som prosedyre! (forts.)

```
(define (cons x y)
  (lambda (message)
    (cond ((= message 0) x)
          ((= message 1) y))))

(define (car proc)
  (proc 0))

(define (cdr proc)
  (proc 1))

? (cons 1 2)
→ #<procedure>

? (cdr (cons 1 2))
→ 2

? (car (cons 1 (cons 2 3)))
→ 1
```

- Prosedyren som `cons` returnerer brukes bare som en container for å huske parameterbindingene:
- car- og cdr-verdiene (`x` og `y`) 'gjemmes' via såkalt innkapsling:
- *Closures* i vanlig Lisp-sjargong, men ikke i SICP.
- Via prosedyren som returneres av `cons` får vi tilgang til variablene *utenfor* det som ellers ville være rekkevidden av dens binding (prosedyrekroppen til `cons`)!
- Eksempel på konseptuelt interessant teknikk vi skal jobbe mye med senere!

Par som prosedyre: versjon 2.

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

(define (car p)
  ...)

(define (cdr p)
  ...)

? (define foo (cons 42 '()))

? (car foo)
→ 42

? (cdr foo)
→ '()
```

Par som prosedyre: versjon 2.

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

(define (car p)
  ...)

(define (cdr p)
  ...)

? (define foo (cons 42 '()))

? (car foo)
→ 42

? (cdr foo)
→ '()
```

- Å skrive ferdig `car` og `cdr` her blir del av oblig 2a.

Par som prosedyre: versjon 2.

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

(define (car p)
  ...)

(define (cdr p)
  ...)

? (define foo (cons 42 '()))

? (car foo)
→ 42

? (cdr foo)
→ '()
```

- Å skrive ferdig `car` og `cdr` her blir del av oblig 2a.
- Prosedyrerepresentasjonene våre er mindre effektive enn vanlige par.
- Men gode eksempler på hvor mye man kan gjøre med bare prosedyrer.
- Senere i kurset skal vi se flere praktiske eksempler på prosedyrer som datastrukturer!

Lister av lister

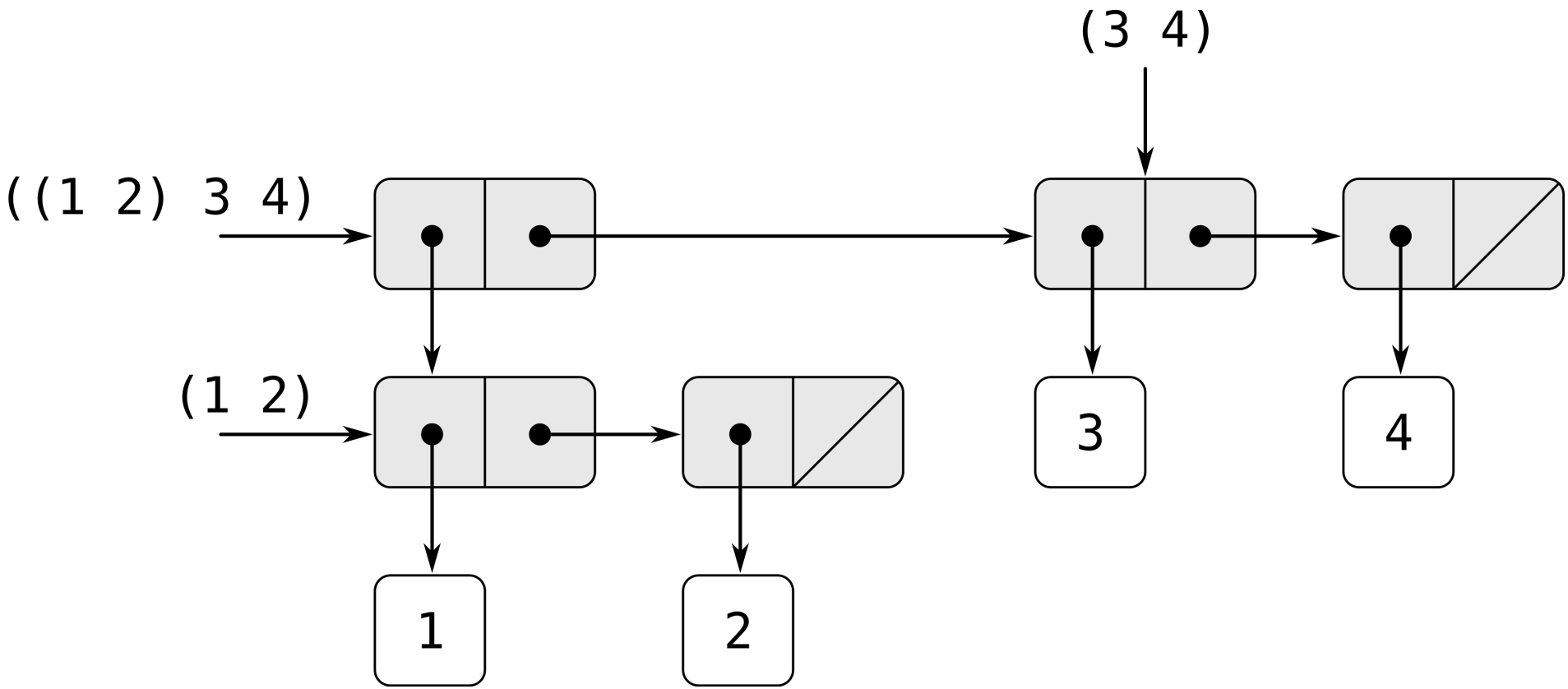
- *Closure property* i SICP: Liste-elementer kan selv være lister . . .
- Som igjen kan bestå av nye lister.

```
? (list (list 1 2) 3 4)
→ ((1 2) 3 4)
```

Lister av lister

- *Closure property* i SICP: Liste-elementer kan selv være lister . . .
- Som igjen kan bestå av nye lister.

```
? (list (list 1 2) 3 4)
→ ((1 2) 3 4)
```



Trær

- *Lister* har så langt representert sekvenser.
- *Lister av lister* kan sees som trær.
- Nyttig for å representere hierarkisk strukturerte data.



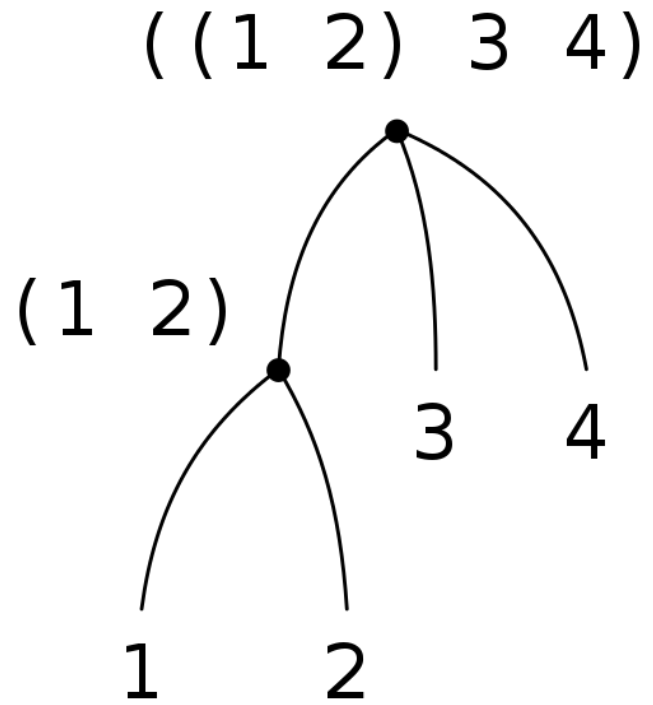
Litt terminologi for trær

- Et *tre* er en type graf, en mengde *noder* forbundet med *kanter*.
- Den øverste noden kalles rot.
- Interne noder har barn / døtre: nye subtrær.
- Løvnoder har ikke barn (såkalt terminalnoder).
- Noder har kun én forelder.
- Kantene kan ikke være sykliske.
- Binærtrær: nodene har maks to døtre, venstre/høyre.



Lister som trær

- Flate lister: sekvenser.
- Lister av lister: kan sees som trær.
  - Elementer som selv er lister er subtrær.
  - Løvnodene i treet er de atomære elementene som ikke er lister.
- PS: Har verdier kun på løvnodene her; skal etterhvert lage en mer generell implementasjon av trær.



Rekursjon på trær (eksempel 1:3)



- Eksempel: telle løvnoder (parallelt til `length` på sekvenser).

```
? (count-leaves '((1 2) 3 4)) → 4
```

Rekursjon på trær (eksempel 1:3)



- Eksempel: telle løvnoder (parallelt til `length` på sekvenser).

```
? (count-leaves '((1 2) 3 4)) → 4
```

- Må passe på at rekursjonen går ned i hver (element)liste.
- Må tenke på tre forskjellige situasjoner. Argumentet kan være:
  - den tomme lista
  - en `cons`-celle (et tre)
  - noe annet (løvnode)

Rekursjon på trær (eksempel 1:3)



- Eksempel: telle løvnoder (parallelt til `length` på sekvenser).

```
? (count-leaves '((1 2) 3 4)) → 4
```

- Må passe på at rekursjonen går ned i hver (element)liste.
- Må tenke på tre forskjellige situasjoner. Argumentet kan være:
  - den tomme lista
  - en cons-celle (et tre)
  - noe annet (løvnode)

```
(define (count-leaves tree)
  (cond ((null? tree) 0)                    ;; ikke løvnode
        ((pair? tree)
         (+ (count-leaves (car tree))
            (count-leaves (cdr tree))))
        (else 1)))                          ;; løvnode
```

Rekursjon på trær (eksempel 2:3)

```
? (fringe '((1 2) ((3) 4)))
→ (1 2 3 4)
```

- Et annet eksempel: samle opp alle løvnodene i en flat liste.

Rekursjon på trær (eksempel 2:3)

```
? (fringe '((1 2) ((3) 4)))
→ (1 2 3 4)

(define (fringe tree)
  (cond ((null? tree) '())
        ((pair? tree)
         (append (fringe (car tree))
                 (fringe (cdr tree))))
        (else (list tree))))
```

- Et annet eksempel: samle opp alle løvnodene i en flat liste.
- Typisk eksempel på rekursjon på lister av lister:
- Vi ønsker å gjøre noe for hvert atomære element (*løvnodene*).
- Må passe på at rekursjonen går ned i hver liste.
- To basistilfeller: tomt tre eller løvnode.

Rekursjon på trær (eksempel 2:3)

```
? (fringe '((1 2) ((3) 4)))
→ (1 2 3 4)

(define (fringe tree)
  (cond ((null? tree) '())
        ((pair? tree)
         (append (fringe (car tree))
                 (fringe (cdr tree))))
        (else (list tree))))


;; append kombinerer to lister;
;; sammenlikning med cons:

? (append '(1 2) '(3)) → (1 2 3)

? (cons '(1 2) '(3)) →
```

- Et annet eksempel: samle opp alle løvnodene i en flat liste.
- Typisk eksempel på rekursjon på lister av lister:
- Vi ønsker å gjøre noe for hvert atomære element (*løvnodene*).
- Må passe på at rekursjonen går ned i hver liste.
- To basistilfeller: tomt tre eller løvnode.

Rekursjon på trær (eksempel 2:3)

```
? (fringe '((1 2) ((3) 4)))
→ (1 2 3 4)

(define (fringe tree)
  (cond ((null? tree) '())
        ((pair? tree)
         (append (fringe (car tree))
                 (fringe (cdr tree))))
        (else (list tree))))


;; append kombinerer to lister;
;; sammenlikning med cons:

? (append '(1 2) '(3)) → (1 2 3)

? (cons '(1 2) '(3)) → ((1 2) 3)
```

- Et annet eksempel: samle opp alle løvnodene i en flat liste.
- Typisk eksempel på rekursjon på lister av lister:
- Vi ønsker å gjøre noe for hvert atomære element (*løvnodene*).
- Må passe på at rekursjonen går ned i hver liste.
- To basistilfeller: tomt tre eller løvnode.

Rekursjon på trær (eksempel 3:3)



- `map` er definert for lister: `tree-map` kan defineres for lister av lister:

```
? (tree-map square '((1 2) 3 4)) → ((1 4) 9 16)
```

Rekursjon på trær (eksempel 3:3)



- `map` er definert for lister: `tree-map` kan defineres for lister av lister:

```
? (tree-map square '((1 2) 3 4)) → ((1 4) 9 16)
```

```
(define (tree-map proc tree)
  (cond ((null? tree) '())
        ((pair? tree)
         (cons (tree-map proc (car tree))
               (tree-map proc (cdr tree))))
        (else (proc tree))))
```

Rekursjon på trær (eksempel 3:3)



- `map` er definert for lister: `tree-map` kan defineres for lister av lister:

```
? (tree-map square '((1 2) 3 4)) → ((1 4) 9 16)
```

```
(define (tree-map proc tree)
  (cond ((null? tree) '())
        ((pair? tree)
         (cons (tree-map proc (car tree))
               (tree-map proc (cdr tree))))
        (else (proc tree))))
```

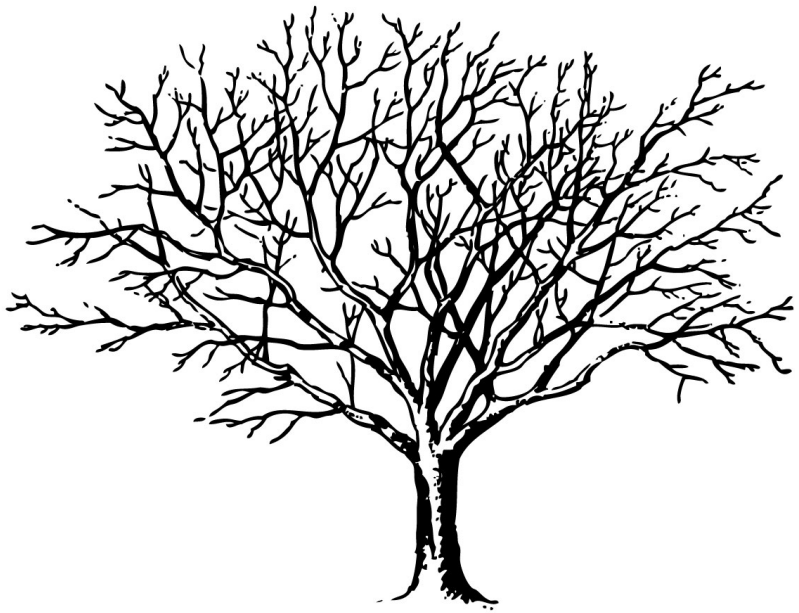
- Alternativt kan `map` brukes, i kombinasjon med rekursjon:

```
(define (tree-map proc tree)
  (map (lambda (subtree)
         (if (pair? subtree) (tree-map proc subtree) (proc subtree)))
       tree))
```

- Hvorfor trengs det bare én rekursiv oppkalling i denne varianten?

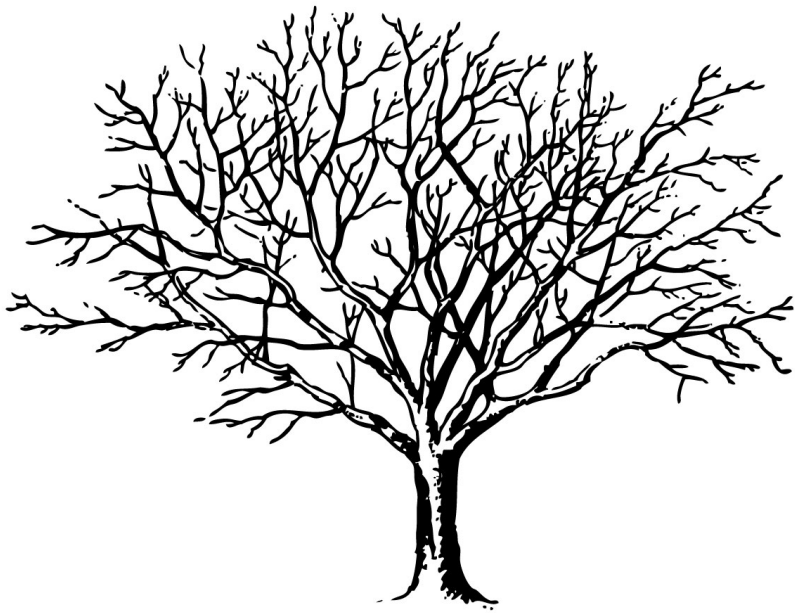
Trerekursjon

- `count-leaves`, `fringe`, osv. vil gi en trerekursiv prosess.
- Fikk dårlig rykte da vi så på `fib` tidligere:
  eksponentiell vekst og masse redundante beregninger.
- Prosesstreet gjenspeilet trinnene i en naiv algoritme.



Trerekursjon

- `count-leaves`, `fringe`, osv. vil gi en trerekursiv prosess.
- Fikk dårlig rykte da vi så på `fib` tidligere:
  eksponentiell vekst og masse redundante beregninger.
- Prosesstreet gjenspeilet trinnene i en naiv algoritme.
- Her gjenspeiler det bare strukturen til inputdata.
- Kompleksiteten lineær i antall noder i input. Ingen redundans.



Veien videre

- Så langt: Kan behandle nøstede lister som trær,
- men bruker bare `pair?`, `null?`, `cons`, `car` og `cdr` direkte.

Veien videre

- Så langt: Kan behandle nøstede lister som trær,
- men bruker bare `pair?`, `null?`, `cons`, `car` og `cdr` direkte.
- Ingen abstraksjon!
- Vi skal definere abstrakte datatyper for å representere trær.
- Ikke bare *løvnodene*, men også de *interne nodene* kan ha verdier.
- Skal også jobbe med eksempler der rekursjonen velger å kun følge én gren (i stedet for å gå ned i alle subtrær).

Veien videre

- Så langt: Kan behandle nøstede lister som trær,
- men bruker bare `pair?`, `null?`, `cons`, `car` og `cdr` direkte.
- Ingen abstraksjon!
- Vi skal definere abstrakte datatyper for å representere trær.
- Ikke bare *løvnodene*, men også de *interne nodene* kan ha verdier.
- Skal også jobbe med eksempler der rekursjonen velger å kun følge én gren (i stedet for å gå ned i alle subtrær).
- Som praktiske eksempler skal vi se på hvordan både mengder og komprimeringskoder kan implementeres effektivt som trær.

Neste uke(r)

- Mer om både dataabstraksjon og hierarkiske strukturer
- Praktisk eksempel 1: representasjon av mengder.
- Praktisk eksempel 2: Huffmankoding (hovedtema for oblig 2a).

