

IN2040: Funksjonell Programmering

Oppsummering og eksamensforberedelser

Martin Steffem

Universitetet i Oslo

13. uke (15. Nov. 2024)



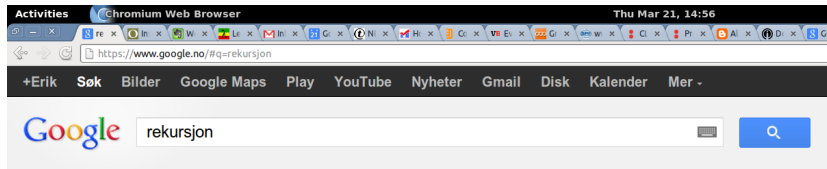
- ▶ Kort oppsummering
- ▶ Praktisk om eksamen
- ▶ Prøveeksamen: Spørsmål og svar.



- ▶ Abstraksjon
- ▶ Rekursjon
- ▶ Prosedyrer
- ▶ Evaluering
- ▶ Funksjonell programmering



- ▶ Abstraksjon for å kontrollere kompleksitet (skjuler og isolerer).
- ▶ Kan se på prosedyredefinisjoner 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 ulike 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; bygget på lambda.
- ▶ Metalingvistisk abstraksjon



Nettet

Bilder

Google Maps

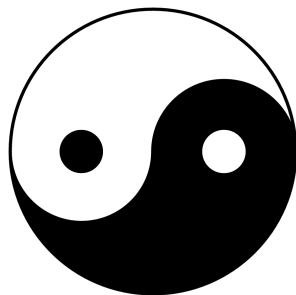
Mer ▼

Innstillinger for søket

Omtrent 13 900 resultater (0,19 sekunder)

Mente du: [rekursjon](#)

- ▶ Rekursive definisjoner av prosedyrer
 - ▶ Direkte rekursjon og gjensidig rekursjon
- ▶ Prosedyre vs prosess
 - ▶ Rekursive og trerekursive prosesser
 - ▶ Iterative prosesser
 - ▶ Kompleksitet
- ▶ Ulike typer idiomer for rekursjon, f.eks;
 - ▶ over tall, over lister, over trær, ... inn / ut,
 - ▶ basistilfeller,
 - ▶ utsatt evaluering
 - ▶ halerekursjon, m.m.
- ▶ 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



- ▶ Prosedyrer som 1.klasses objekter
- ▶ Høyereordens prosedyrer:
andre prosedyrer som argumenter / returverdi
- ▶ Lambda-uttrykk; navngitte og anonyme prosedyrer
- ▶ let og lokale variabler via anonyme prosedyrer
- ▶ 'Let-over-lambda'
- ▶ Prosedyrebasert objektorientering
- ▶ + lokal tilstand via innkapsling + verditilordning
- ▶ Prosedyrer som datastrukturer (f.eks cons-celler)

A

- ▶ Evalueringsregler
- ▶ Substitusjonsmodellen $\Leftrightarrow$ omgivelsesmodellen
- ▶ Metasirkulær evaluator
- ▶ Kode som data
- ▶ Utsatt evaluering
- ▶ Memoisering
- ▶ Eager/Applicative-order vs Lazy/normal-order evaluation
- ▶ Macro-definisjoner for å definere special forms
- ▶ Samspillet mellom (ikke-)funksjonell programmering og ulike modeller og strategier for evaluering.



- ▶ Hovedfokus: Funksjonell programmering.
- ▶ Prosedyrer som spesifikasjon 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).
 - ▶ Prosedyrer kalles for sin returverdi: ingen side-effekter.
 - ▶ Semantikken til uttrykk uavhengig av hvor og når de brukes.
- ▶ La så til muterbare datastrukturer og destruktive operasjoner for modellering av endringer i tid og lokal tilstand.

 λ $+$ $!$

IN2040: Funksjonell Programmering

Eksamensforberedelser

Martin Steffen

Universitetet i Oslo

13. uke (20. Nov. 2023)



- ▶ Hvem sanker inn flest oblig-poeng igjennom semesteret?
- ▶ Vinnerene får tradisjonen tro en rekursiv t-skjorte!
- ▶ kommer senere per epost + websider



- ▶ **FUI** sender etterhvert ut invitasjoner til **kursevaluering**.
- ▶ Det betyr mye at dere svarer!
- ▶ Midtveisevalueringen er bare til kursinternt bruk:
- ▶ FUIs kursevaluering er mer omfattende og synlig på instituttnivå.



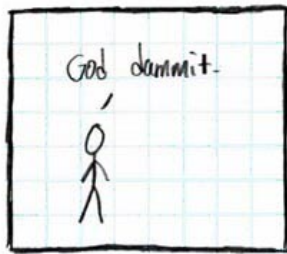
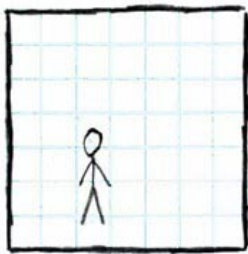
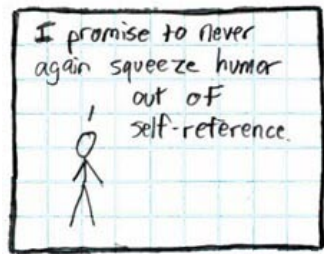
- ▶ vanlig eksamen (ikke **hjemmeeksamen** som under Corona)
- ▶ **8. desember kl. 09:00** (4 timer) fra SICP, se oversikt på semestersiden:
<https://www.uio.no/studier/emner/matnat/ifi/IN2040/h23/detaljert-pensumliste.html>
- ▶ **NB:** Sjekk også i StudentWeb at du faktisk er oppmeldt!
- ▶ **Tidligere eksamensoppgaver** fortsatt representative:

<https://github.uio.no/IN2040/h23/tree/main/eksamensoppgaver>



Tusen takk!

Anton Zelentsov, Jakub Mikolaj Wasylkow, Sergey Jakobsen, Maham Iftikhar, Ole Jørgen Nordstrand, Elise Møllerstedt Gunnestad, Magnus Økstad, Oskar Haukebøe, Lise Chen, Mathias Strømberg Durkis, Emir Kukuruzovicm, Dan Ngyen



<https://xkcd.com/33/>

IN2040: Funksjonell Programmering

Eksamensforberedelser + kommentarer til eksamen 2023

Martin Steffen

Universitetet i Oslo

13. uke (15. Nov. 2024)



1) Små spørsmål



- ▶ *prosedyreobjekt?*
- ▶ Hva returneres (hvis noe) på den siste linjen:

```
(define (x) x)  
(x)           ;; -> ??
```

```
(define (x y . z) (cons y z))  
(x 1 2 3 4)    ;; -> ??
```

2.1) Listestrukturer (ingen strukturdeling, ikke sirkulær)



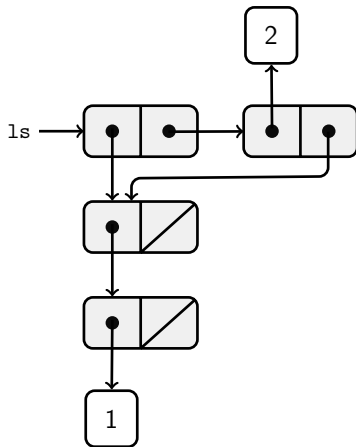
```
(list? 4)                ;; -> #f
(list? '())              ;; -> #t
(list? '(1 2))           ;; -> #t
(list? '(1 . 2))         ;; -> #f
(list? '(1 2 (2 3)))     ;; -> #t
(list? '((1 2) (2 . 3))) ;; -> #t
```

2.2) count-cons-cells



```
( count-cons-cells '())           ;; -> 0  
( count-cons-cells '(1 . 2))      ;; -> 1  
( count-cons-cells '(1 2 3))      ;; -> 3  
( count-cons-cells '(1 (2 3)))    ;; -> 4
```

2.3) Strukturdeling



- **uten** destruktive operasjoner

2.4) komplekse listestrukturer



- ▶ `list?` kalt med strukturen `ls` fra forrige deloppgave?
- ▶ **sirkulær?**

```
(define foo '(1 2 3))  
(set-cdr! (cdr (cdr foo)) (cdr foo))  
foo          ;; -> (1 . #0=(2 3 . #0))  
  
(list? foo)  ;; -> #f
```

3) split-list



```
(splitlist odd? '(1 2 3 4 5 6 7 8)) ;; -> ((1 3 5 7) 2 4 6 8)
(splitlist odd? '())                ;; -> (())
```

4) Sekvensoperasjoner: reduce



Listing 1: reduce

```
(define reduce ;; aka fold, fold-right, accumulate ...  
  (lambda (proc init items)  
    (if (null? items)  
        init  
        (proc (car items)  
              (reduce proc init (cdr items))))))
```

4.1) reduce-left



```
(+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))
```

eller

```
1 + (2 + (3 + (4 + (5 + 0))))
```

Programmer reduce-left

```
(+ (+ (+ (+ (+ 0 1) 2) 3) 4) 5) ;; -> 15
```



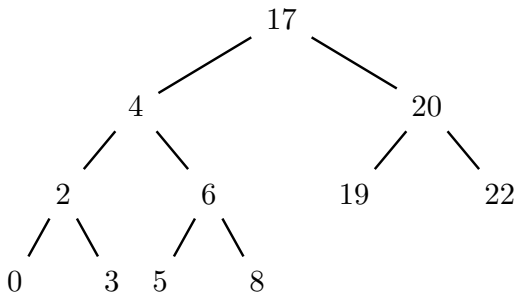
```
(reduce-left cons '() '(1 2 3)) ;; → ???  
(reduce      cons '() '(1 2 3)) ;; → ???
```

5) Trær som abstrakt datatype



```
(make-tree entry left right) ;; creates a tree
(make-leaf entry)           ;; creates a leaf
(entry tree)                ;; selector: extract node
(left-branch tree)          ;; selector: subtree
(right-branch tree)         ;; selector: subtree
(leaf? tree)                ;; predicate: is it a leaf
```

5) Ordnet



Oppgave

Er et gitt tre er **ordnet** eller ikke

6.1) stream-iterate



Antar **grensesnitt**

```
(define s
  (stream-iterate (lambda (x) (+ 1 x)) 42))

(show-stream s 5)           ;;  $\rightarrow$  42 43 44 45 46 ...

(define s2
  (stream-iterate (lambda (x) (* 2 x)) 1))

(show-stream s2 5)          ;;  $\rightarrow$  1 2 4 8 16 ...
```

6.2) Strømmer og halerekursjon



```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))
```

► halerekursjon: vanlig motivasjonen?

fungerer det?

```
(define (integers-starting-from n)
  (define (iter n acc)
    (iter (+ 1 n) (cons-stream n acc)))
  (iter n the-empty-stream))

(define nats (integers-starting-from 0))
```

```
(define rodekors (make-charity)) ;; create one charity account

(define peter (rodekors))          ;; proc.-based object "peter"

(peter 'status)                    ;; → (0 . 0)
((peter 'donate) 10)                ;; → 10
((peter 'donate) 40)                ;; → 50

(define paul (rodekors))           ;; another "donor" called paul

(paul 'status)                     ;; → (0 . 50)
((paul 'donate) 15)                 ;; → 15
(paul 'status)                     ;; → (15 . 65)
(peter 'status)                     ;; → (50 . 65)

(define legerutengrenser (make-charity)) ;; a second charity

(define mary (legerutengrenser))

((mary 'donate) 33)                 ;; → 33
(mary 'status)                      ;; → (33 . 33)
(peter 'status)                      ;; → (50 . 65)
```

IN2040: Funksjonell Programmering

*Eksamensforberedelser + kommentarer til eksamen
2022-b*

Martin Steffen

Universitetet i Oslo

13. uke (15. Nov. 2024)



1. prosedyrer som førsteklasses objekter
2. evaleringsstrategier
3. muterbare strukturer

```
(define (f x y)
  (if (= x 0)
      1
      (f (- x 1) (f (- x y) y))))
(f 1 0) ;; ???
```

1 Konsepter c: muterbare lister



- ▶ `set!` og `set-car!`
- ▶ rekkevide (scope) for `elem`

Se her:

```
(define (foo1 elems)
  (define (bar1)
    (set-car! elems 42)
    (set! elems 17))
  (bar1)
  elems)
```

```
(define (foo2 elems)
  (define (bar2 elems)
    (set-car! elems 42)
    (set! elems 17))
  (bar2 elems)
  elems)
```

- ▶ ordnære lister
- ▶ listrekursjon

Kalleeksempler

(remove-first even? '())	:: → ()
(remove-first even? '(1 2 3 4))	:: → (1 3 4)
(remove-first even? '(1 3 5))	:: → (1 3 5)

3: Listestrukturer: atom?



(atom? 15)	:: → #t
(atom? 'abc)	:: → #t
(atom? '())	:: → #f
(atom? '(1))	:: → #f
(atom? '(1 (2 3)))	:: → #f
(atom? '(1 . 2))	:: → #f

```
(flatten '())                ;; → ()  
(flatten '(1 2 3))          ;; → (1 2 3)  
(flatten '(1 (2 3)))        ;; → (1 2 3)  
(flatten '((1) (2 (3 4) 5) (6 7) 8))  
;; → (1 2 3 4 5 6 7 8)
```

```
(map (lambda (x) (* x x)) '())      ;; → ()  
(map (lambda (x) (* x x)) '(1 2 3)) ;; → (1 4 9)
```

4a Høyereordens prosedyrer: repeat



```
(repeat (lambda (x) (* x 2)) 0 5) ;; → 5  
(repeat (lambda (x) (* x 2)) 1 5) ;; → 10  
(repeat (lambda (x) (* x 2)) 2 5) ;; → 20  
(repeat (lambda (x) (* x 2)) 3 5) ;; → 40
```

```
(repeat2 square 3)
;; → #<procedure>
((repeat2 square 3) 2)                ;; → 256
((repeat2 (lambda (x) (* x 2)) 3) 5)  ;; → 40
```

- ▶ litt mer “høyre-orden”
- ▶ base-case?



```
(define (plus x y) (+ x y))    ;; addition with 2 arguments
```

```
(define plus-curried (curry plus))
```

```
(plus 1 2)                    ;; → 3
```

```
((plus-curried 1) 2)         ;; → 3
```

- ▶ funksjonell vs imperative
- ▶ når stopper rekursjon

imperative

```
mylist                ;; → #0=(1 2 3 . #0#)
(cdr (cdr (cdr mylist))) ;; → #0=(1 2 3 . #0#)
(undo-cycle-imp mylist)
mylist                ;; → (1 2 3)
```

► funksjonell

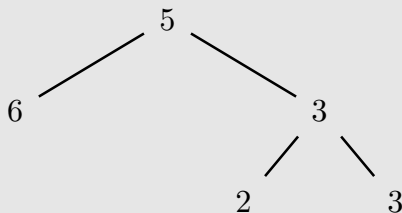
imperative

mylist	;; → #0=(1 2 3 . #0#)
(undo-cycle-fun mylist)	;; → (1 2 3)
mylist	;; → #0=(1 2 3 . #0#)

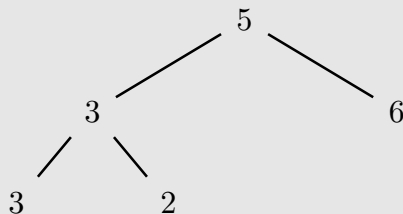
- ▶ ikke (umiddelbart) nøstete lister
- ▶ egen grensesnitt

```
(make-tree entry left right) ;; creates a tree
(make-leaf entry)           ;; creates a leaf
(entry tree)                ;; selector: extract node's value
(left-branch tree)          ;; selector: subtree
(right-branch tree)         ;; selector: subtree
(leaf? tree)                ;; predicate: is it a leaf?
```

input



output

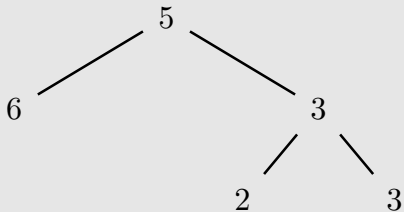


- ▶ tre-rekursjon (og rekursjon over trær)
- ▶ basis og rekursjons fall?

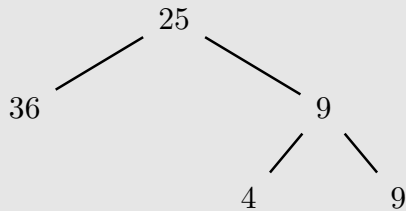
3 Trær ADT b: tree-map



input



output



6: “Flette sammen” strømmer (stream-merge) a+b



```
;;; sn: stream of nats  $\geq 1$ 
;;; sq stream of squares of nats  $\geq 1$ 
(show-stream sn 10)    ;;  $\rightarrow$  1 2 3 4 5 6 7 8 9 10 ...
(show-stream sq 10)    ;;  $\rightarrow$  1 4 9 16 25 36 49 64 81 100 ...

(show-stream
 (stream-merge sn sq) 13) ;;  $\rightarrow$  1 1 2 3 4 4 5 6 7 8 9 9 10 ...
```

► kan man **sortere** strømmer?

- Cf: **køer** (de var uten OO)

```
(define mystack (make-empty-stack))  
(empty? mystack)           ;; → #t  
(push mystack 1)  
(push mystack 2)  
(push mystack 3)  
(empty? mystack)           ;; → #f  
(pop mystack)               ;; → 3  
(pop mystack)               ;; → 2  
(pop mystack)               ;; → 1  
(empty? mystack)           ;; → #t  
(pop mystack)               ;; → "empty stack!"
```

IN2040: Funksjonell Programmering

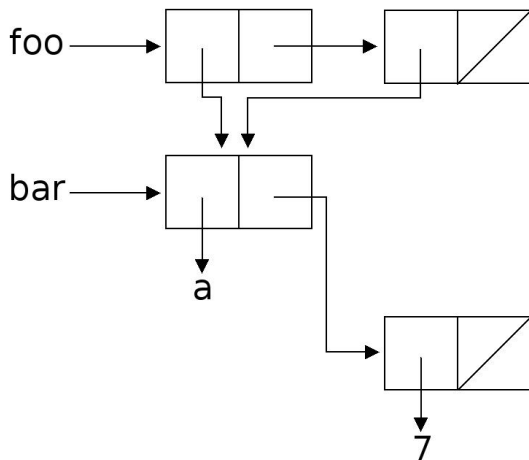
Kommentarer til prøveeksamen

Martin Steffen

Universitetet i Oslo



1+2: Grunnleggende (6 poeng)



```
? (define foo '(a b))  
?  
? (define bar foo)  
?  
? (set! foo '(c d))  
?  
? (set-car! foo bar)  
?  
? (set-car! (cdr foo)  
      (car foo))  
?  
? (set-cdr! bar  
      (list 7))  
?  
? foo → ((a 7) (a 7))
```

3: Mysterium (6 poeng)



```
(define (x y z)
  (or (null? z)
      (and (y (car z))
            (x y (cdr z)))))
```

- `x` er en høyereordens prosedyre (eller mer spesifikt et predikat) som tar et annet predikat og en liste som argument og returnerer `#f` dersom et element i lista tester usant for predikatet, og `#t` ellers. Eksempel:

```
? (x odd? '(1 3 5 7 9)) → #t
```

4: Rekursjon (8 poeng)



Prekode

```
? (define (avg first . rest)
  (let ((arguments (cons first rest)))
    (/ (apply + arguments)
       (length arguments))))
```

```
? (avg 1 2 3) → 2
```

```
? (avg 1 1 2 2 3 3) → 2
```

Eksempel på rekursiv variant

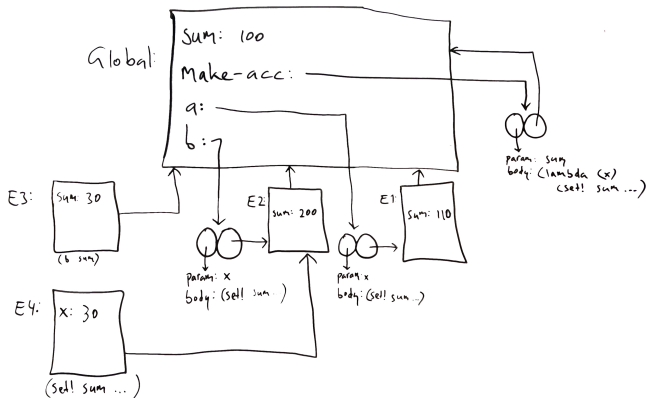
```
(define (avg first . rest)
  (define (iter sum count numbers)
    (if (null? numbers)
        (/ sum count)
        (iter (+ sum (car numbers))
              (+ 1 count)
              (cdr numbers))))
  (iter first 1 rest))
```

5: Omgivelser (10 poeng)



```
? (define sum 100)
? (define (make-acc sum)
  (lambda (x)
    (set! sum (+ sum x))
    sum))
```

```
? (define a (make-acc sum))
? (define b (make-acc 200))
? (a 10)
? (let ((sum 30))
  (b sum))
```



Funksjonell (og rekursiv)

```
(define (transform-if test trans seq)
  (if (null? seq)
      '()
      (cons (if (test (car seq))
                 (trans (car seq))
                 (car seq))
            (transform-if test trans (cdr seq)))))
```

- **Funksjonell**: lager en helt ny liste.
- **Destruktiv**: Lager ingen nye cons-celler men gjenbraker bare den eksisterende strukturen.

Destruktiv (og halerekursiv)

```
(define (transform-if! test trans seq)
  (define (iter sub)
    (cond ((null? sub) seq)
          ((test (car sub))
           (set-car! sub (trans (car sub)))
           (iter (cdr sub)))
          (else (iter (cdr sub)))))
  (iter seq))
```

8: Evalueringsstrategier (16 poeng)



Eager / applicative-order

- ▶ Argumentuttrykkene evalueres *før* en prosedyre anvendes.
- ▶ **Standardstrategien** ved evaluering av prosedyrekall i Scheme.
 - (Kalles også noen ganger for *call-by-value* eller *strict evaluation*).

Lazy / normal-order

- ▶ Argumentuttrykkene evalueres *først* ved behov.
- ▶ (Kalles også *non-strict evaluation*.)
- ▶ Brukes i Scheme ved evaluering av **special-forms** som f.eks *if* og *and*.
- ▶ I noen varianter av *normal-order* re-evalueres uttrykkene hver gang de brukes (*call-by-name*). Men *lazy evaluation* inkluderer gjerne også **memoisering**: Argumenter evalueres da maks. én gang (*call-by-need*).
- ▶ I kap. 4 så vi hvordan den **metasirkulære evaluatoren** kunne endres til å implementere en Scheme-variant med *lazy evaluation*.
- ▶ Har også sett hvordan vi i vanlig Scheme kan jobbe med **strømmer** via **delay**, en innebygd *special-form* som lar oss eksplisitt velge å utsette evaluering av uttrykk.
- ▶ (Egner seg best for funksjonelle språk; kan være forvirrende ifb prosedyrer som har side-effekter siden det kan være vanskelig å forutsi rekkefølgen på operasjoner.)



9: same-fringe?



Prekode

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

Eksempler

```
? (same-fringe?
   '((1 2) 3 4)
   '((1 (2)) 3 (4)))
=)
```

→ #t

```
? (same-fringe?
   '((1 2) 3 4)
   '((1 7) 3 4))
=)
```

→ #f

```
(define (same-fringe? tree1 tree2 same?)
  (define (iter t1 t2)
    (cond ((and (null? t1) (null? t2)) #t)
          ((or (null? t1) (null? t2)) #f)
          ((not (same? (car t1) (car t2))) #f)
          (else (iter (cdr t1) (cdr t2)))))
  (iter (fringe tree1) (fringe tree2)))
```

```
(define (fringe-stream tree)
  (cond ((null? tree) the-empty-stream)
        ((pair? (car tree))
         (stream-append (fringe-stream (car tree))
                        (fringe-stream (cdr tree))))
        (else (cons-stream (car tree)
                            (fringe-stream (cdr tree))))))
```

```
(define (same-fringe-stream? tree1 tree2 same?)
  (define (iter t1 t2)
    (cond ((and (stream-null? t1)
                (stream-null? t2)) #t)
          ((or (stream-null? t1)
               (stream-null? t2)) #f)
          ((not (same? (stream-car t1)
                      (stream-car t2))) #f)
          (else (iter (stream-cdr t1)
                      (stream-cdr t2)))))
  (iter (fringe-stream tree1) (fringe-stream tree2)))
```

- *Hvor mange cons-celler genereres (gitt implementasjonen over)?*

```
? (fringe-stream '(1 ((2) 3 4)))
```

```
? (fringe-stream '((1 2) 3 4))
```

- De to uttrykkene genererer **henholdsvis 1 og 3 cons-celler** (selv om returverdiene på REPLet tilsynelatende vil være identiske).
- I **uttrykk #1** er det den siste cond-grenen som slår til og vi får umiddelbart et kall på cons-stream; en *special form* som gir utsatt evaluering av cdr-delen.
- I **uttrykk #2** får vi i stedet et kall på prosedyren stream-append; dette vil i seg selv gi én cons, men i tillegg vil kallene på fringe-stream i begge argumentuttrykkene også gi én cons hver.

- ▶ *I hvilke tilfeller vil det være en fordel å bruke den siste strømbaserte varianten fremfor den første? Gi en kort begrunnelse.*
- ▶ Dersom sekvensene av løvnoder faktisk er *like* vil strømvarianten ikke gi noen besparelser,
- ▶ men dersom de er *ulike* vil den spare oss for å generere hele sekvensene:
- ▶ Kan avbryte så fort vi finner to ulike løvnoder, og løftet om å generere resten av sekvensen innfris da aldri. Jo større trær jo større besparelser.

Kalleksempler

```
? (define original fringe)

? (set! fringe (monitor fringe))

? (fringe '((1 2) 3)) → (1 2 3)

? (fringe 'count) → 6

? (fringe '((1 2) 3)) → (1 2 3)

? (fringe 'count) → 12

? (fringe 'zero)

? (fringe '((1 2) 3)) → (1 2 3)

? (fringe 'count) → 6

? (set! fringe (fringe 'reset))

? (eq? fringe original) → #t
```

13: Kallstatistikk (20 poeng)



```
(define (monitor procedure)
  (let ((count 0))
    (lambda arguments
      (let ((message (and (not (null? arguments))
                           (car arguments))))
        (cond ((eq? message 'zero) (set! count 0))
              ((eq? message 'count) count)
              ((eq? message 'reset) procedure)
              (else (set! count (+ count 1))
                    (apply procedure arguments)))))))
```

- Ting som må være på plass for å få full pott:
 - kan gjenopprette den opprinnelige prosedyren.
 - kan overvåke vilkårlig mange prosedyrer på en gang.
 - bruker lokal tilstand for å huske opprinnelig prosedyre og teller, altså ikke globale variabler.
 - fungerer for prosedyrer som tar vilkårlig antall argumenter.
 - støtter alle beskjedene (reset / count / zero).