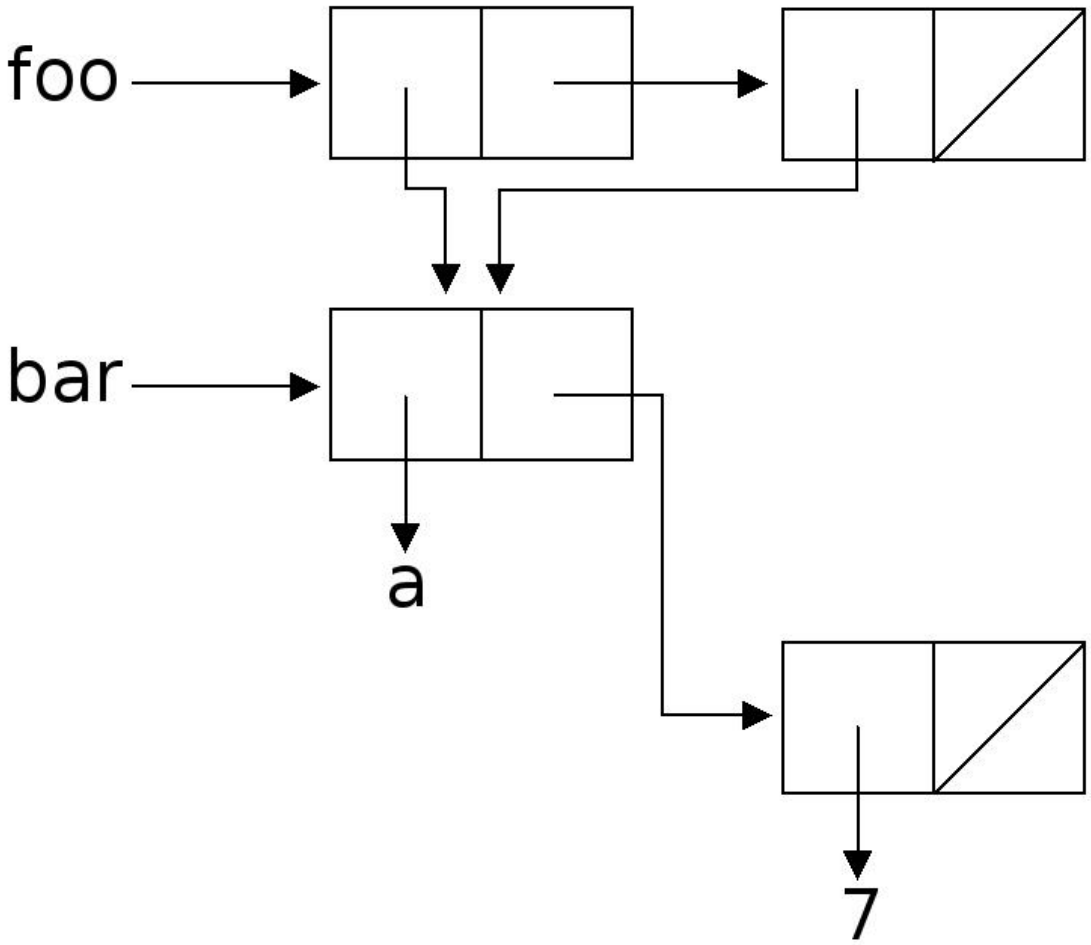
IN2040: Funksjonell Programmering

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
```

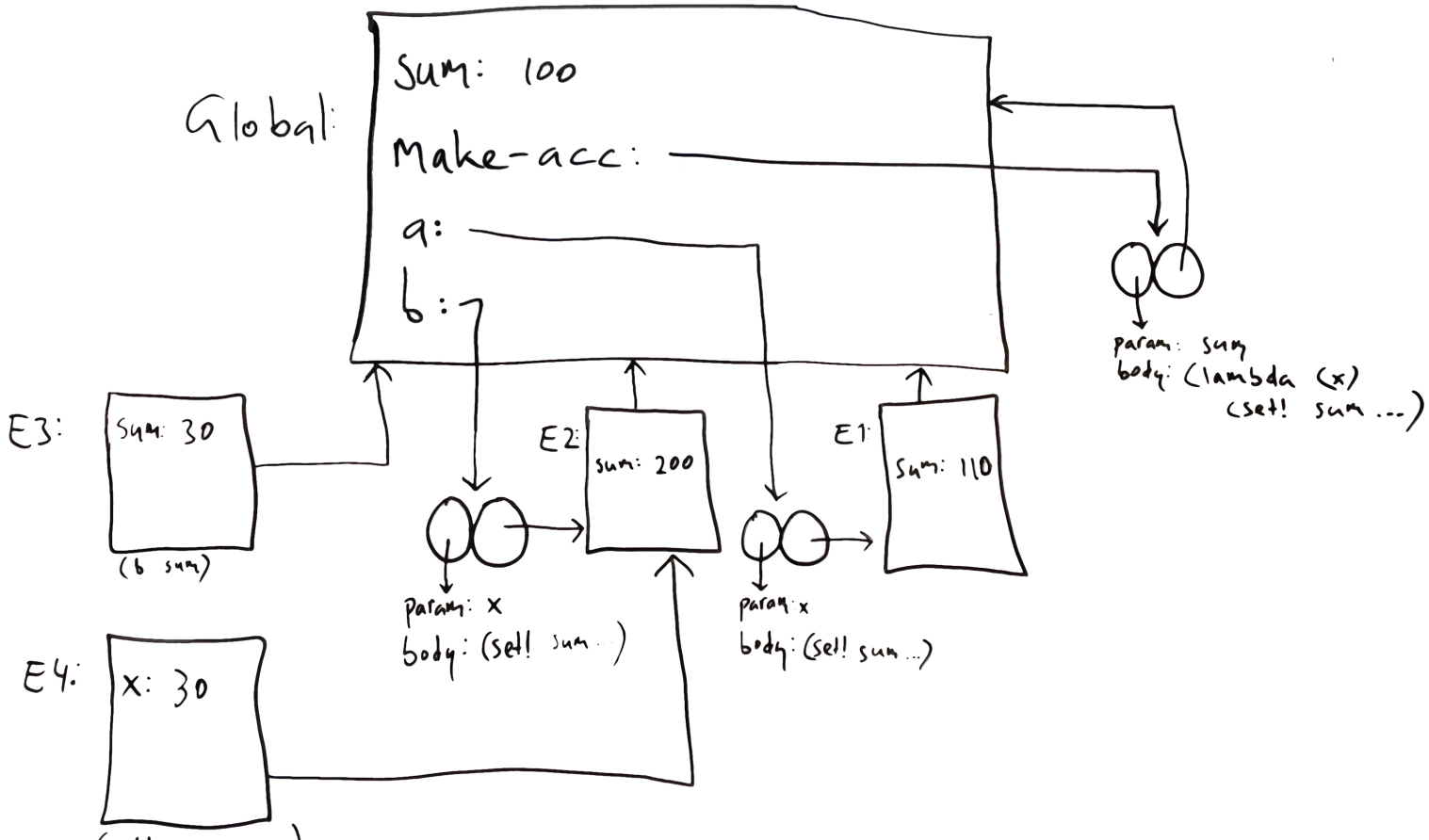
Eksempel på rekursiv variant

```
(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))
```



6+7: Høyereordens prosedyrer (14 poeng)

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)))))
```

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))
```

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

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.)

Dataabstraksjon og strømmer (20 poeng)

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)))
```

10: strømversjoner

```
(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 *u*like 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.

13: Kallstatistikk (20 poeng)

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`).