IN2040: Funksjonell Programmering

Lister og høyereordens prosedyrer

Martin Steffen

Universitetet i Oslo

3. uke (06.09.2022)

Agenda

Forrige uke

- Substitusjonsmodellen og evalueringsstrategier.
- Blokkstruktur
- Rekursjon og halerekursjon
- Funksjon vs prosedyre vs prosess

I dag

- Lister
- Rekursjon på lister
- `quote`
- Høyereordens prosedyrer
- Anonyme prosedyrer og `lambda`-uttrykk

Repetisjon: $n!$ som rekursiv og iterativ prosess

Rekursiv prosess

```
? (define (fac n)
    (if (= n 1)
        1
        (* n (fac (- n 1)))))



? (fac 7)
⇒(* 7 (fac 6))
⇒(* 7 (* 6 (fac 5)))
⇒(* 7 (* 6 (* 5 (fac 4))))
⇒(* 7 (* 6 (* 5 (* 4 (fac 3)))))
⇒(* 7 (* 6 (* 5 (* 4 (* 3 (fac 2))))))
⇒(* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (fac 1)))))))
⇒(* 7 (* 6 (* 5 (* 4 (* 3 (* 2 1))))))
⇒(* 7 (* 6 (* 5 (* 4 (* 3 2)))))
⇒(* 7 (* 6 (* 5 (* 4 6))))
⇒(* 7 (* 6 (* 5 24)))
⇒(* 7 (* 6 120))
⇒(* 7 720)
→5040
```

Iterativ prosess

```
(define (fac n)
  (define (fac-iter prod count)
    (if (> count n)
        prod
        (fac-iter (* count prod)
                  (+ count 1))))
  (fac-iter 1 1))

? (fac 7)
⇒(fac-iter 1 1)
⇒(fac-iter 1 2)
⇒(fac-iter 2 3)
⇒(fac-iter 6 4)
⇒(fac-iter 24 5)
⇒(fac-iter 120 6)
⇒(fac-iter 720 7)
⇒(fac-iter 5040 8)
→5040
```

Lister

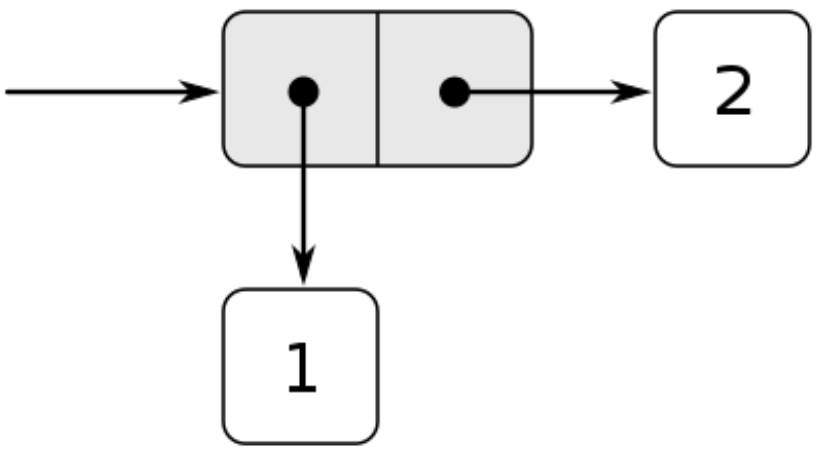
- LISP = LISt Processing
- Hittil har vi sett på lister som prosedyrekall.
- Men lister er også en grunnleggende *datastruktur*.
- For å snakke om lister må vi først snakke om *par*.

Par

- Konstrueres med prosedyren `cons`.
- Par kalles også *cons-celler*.
- En struktur som består av to pekere.
- `car` gir oss den første pekeren.
- `cdr` den andre.

Eksempel

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

? foo → (1 . 2)

? (car foo) → 1

? (cdr foo) → 2
```

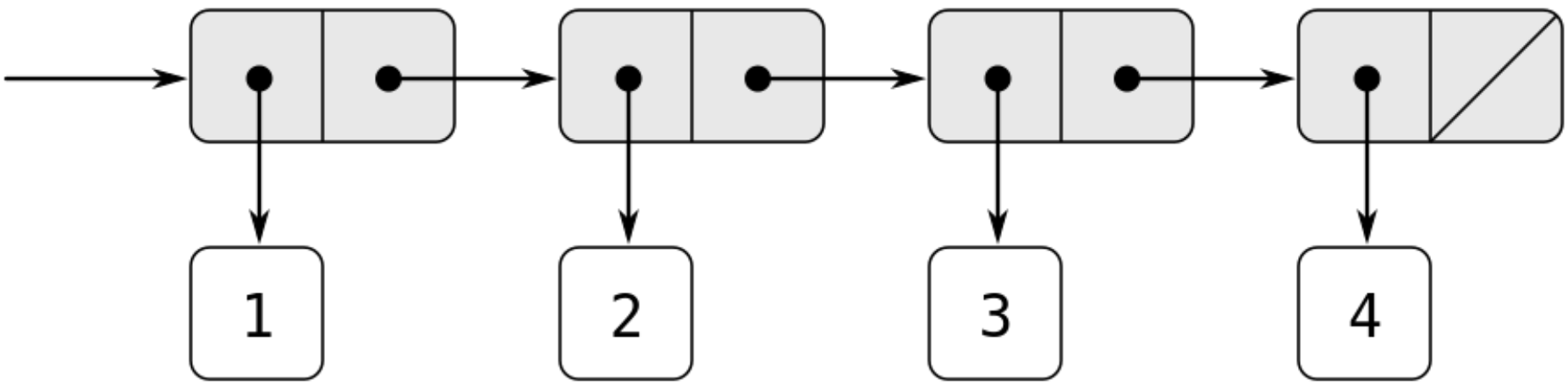


- Kan illustreres med såkalt *boks-og-peker-notasjon*.
- Uttale: "*vi konser 1 på 2*", "*tar kar av foo*" og "*tar kudder av foo*".

Lister

- Lister = kjeder av `cons`-par der siste elementet er den *tomme lista*; `'()`.

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



- Lister kan defineres rekursivt som: `'()` eller et par der `cdr` er en liste.

Listeoperasjoner

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

? (list 1 2 3 4) → (1 2 3 4)

? (define foo (list 1 2 3 4))

? (car foo) → 1

? (cdr foo) → (2 3 4)

? (car (cdr foo)) → 2

? (cdr (cdr (cdr (cdr foo)))) → ()

? (null?  foo) → #f

? (null?  '()) → #t
```

Den tomme lista og quote

- NB: `nil` har ingen verdi i R^{5}RS.
- I stedet brukes `'()` for den tomme lista.
- Skrevet med det vi kaller *quote*.
- En *special form* som gjør at uttrykket som følger ikke evalueres.
- Kortform for `quote`.

```
? '()
→ ()

? ()
→ error:  missing procedure

? '(+ foo bar)
→ (+ foo bar)

? (quote (+ foo bar))
→ (+ foo bar)

? 'foo
→ foo

? (quote foo)
→ foo
```

Flere quote-eksempler

```
? (list 1 2 3 4) → (1 2 3 4)

? '(1 2 3 4) → (1 2 3 4)

? (define foo 42)

? (define bar 8)

? (list foo bar) → (42 8)

? '(foo bar) → (foo bar)

? (+ foo bar) → 50

? (+ 'foo bar) → error:  expected number
```

Rekursjon på lister: Eksempel 1

- Den innebygde prosedyren `length` returnerer lengden av en liste.
- Vi kan skrive den selv, som eksempel på listerekursjon.

```
? (length '(1 2 3)) → 3

? (length '()) → 0

? (length (list)) → 0
```

- Rekursiv plan:
  - Lengden av den tomme lista er 0 (basistilfellet).
  - Lengden av en liste er 1 + lengden av resten av lista uten første element.

Rekursjon på lister: Eksempel 1 (forts.)

- For basistilfellet bruker vi `null?` for å spørre om lista er tom.
- I rekursjonen reduserer vi lista med `cdr`.

```
(define (length items)
  (if (null?  items)
      0
      (+ 1 (length (cdr items)))))
```

- Viser hvordan vi rekursivt traverserer en liste,
- men ofte ønsker vi også å bygge opp en ny samtidig
- (i stedet for å akkumulere en numerisk verdi som her).

Rekursjon på lister: Eksempel 2

```
? (sqr-all '(1 2 3))
→ (1 4 9)

? (sqr-all '(4 8 16))
→ (16 64 256)

? (sqr-all '())
→ ()

(define (sqr-all items)
  (if (null?  items)
      '()
      (cons (square (car items))
            (sqr-all (cdr items)))))
```

- `sqr-all` returnerer en ny liste der hvert element har blitt kvadrert.
- I hvert trinn transformeres det første elementet i lista.
- `cons` samler opp resultatet.
- Rekurserer på `cdr` av lista.
- Basistilfellet gir `'()`.

Rekursjon på lister: Eksempel 3

```
? (abs-all '(-7 2 -3 0))
→ (7 2 3 0)

? (abs-all '(-1 1 -42))
→ (1 1 42)

? (abs-all '())
→ ()

(define (abs-all items)
  (if (null?  items)
      '()
      (cons (abs (car items))
            (abs-all (cdr items)))))
```

- `abs-all` skal returnere en ny liste med absolutt-verdien av hvert element i input-lista.
- Mye av definisjonen til `sqr-all` kan gjenbrukes.
- I hvert trinn transformeres det første elementet i lista.
- `cons` samler opp resultatet.
- Rekurserer på `cdr` av lista.
- Basistilfellet gir `'()`.

Muligheter for generalisering

- `sqr-all` og `abs-all` illustrerer hvordan vi gjør rekursjon over lister.
- Men de er også eksempeler på et generelt mønster:

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

- Når vi har et mønster som går igjen er det tid for abstraksjon!
- Veldig enkelt: vi gjør `proc` til et parameter.
- Denne prosedyren heter vanligvis `map` (og Scheme har en mer generell versjon innebygd).

Høyereordens prosedyrer

```
? (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)

? (map abs '(-1 -2 0 4 -42))
→ (1 2 0 4 42)

? (define (abs-all items)
    (map abs items))
```

- Prosedyrer har førsteklasses status i funksjonelle språk.
- En høyereordens prosedyre er en prosedyre som
  - tar andre prosedyrer som argument, og/eller
  - har andre prosedyrer som sin returverdi.
- Lar oss abstrahere mønstre for hvordan prosedyrer brukes.

λ-uttrykk og anonyme prosedyrer

- Det kan være upraktisk å alltid måtte bruke en forhåndsdefinert og navngitt prosedyre.
- Vi kan også spesifisere prosedyrer direkte med `lambda`-uttrykk.
- Generel form:

  `(lambda (`⟨*argumenter*⟩`)`
  ⟨*kropp*⟩`)`
- `lambda` lager et prosedyreobjekt.
- Hvis vi ikke binder det til et navn med `define` kaller vi det en anonym prosedyre.

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

? (define square
    (lambda (x) (* x x)))

? (square 4)
→ 16

? (lambda (x) (* x x))
→ #<procedure>

? ((lambda (x) (* x x)) 4)
→ 16

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

- Terminologien her avslører røttene...
- Stammer fra lambdakalkylen til Alonzo Church.
- Basis for all funksjonell programmering.
- Var på 30-tallet opptatt av mye av det samme som Alan Turing:
- Å lage en modell for hva beregninger er og hva som kan beregnes.
- Lambdakalkyle: $(\lambda x.\ x \times x)(4)$
- Scheme: `((lambda (x) (* x x)) 4)`

Flere høyereordens prosedyrer

- Mens vi fortsetter å gjøre oss kjent med konseptet høyereordens prosedyrer skal vi se på noen klassiske sekvensoperasjoner:
- `map` har vi allerede sett.
- `reduce` og `filter` er neste.

reduce

- Har snakket om et viktig punkt om å bygge opp et resultat rekursivt:
  - Hva skal returneres når vi når basistilfellet?
  - Avhenger av operasjonen vi bruker for å kombinere resultatene.
- Danner utgangspunktet for en annen klassisk høyereordens prosedyre:

```
? (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

? (reduce * 1 '(1 2 3 4 5)) → 120

? (reduce cons '() '(1 2 3 4 5)) → (1 2 3 4 5)

? (reduce max 0 '(1 2 3 4 5)) → 5
```

reduce

- `reduce` er også kjent som *fold*, *compress*, *accumulate* eller *inject*.
- En måte å tenke på `reduce`:
- Vi beskrev lister som en rekursiv datastruktur i seg selv, bygget opp fra grunnverdien `'()` med kjeder av `cons`-operasjoner.
- `reduce` re-kombinerer elementene med en ny operasjon og grunnverdi.

```
(cons 1 (cons 2 (cons 3 (cons 4 (cons 5  '())))))  → (1 2 3 4 5)

(+    1 (+    2 (+    3 (+    4 (+    5  0))))))   → 15

(*    1 (*    2 (*    3 (*    4 (*    5  1))))))   → 120

(max  1 (max  2 (max  3 (max  4 (max  5  0))))))   → 5
```

Eksempel på bruk av `map` + `reduce`

- Vi skal se på et kort eksempel på bruk av `reduce` sammen med `map`.
- Med vektorer representert som sekvenser (lister).
- Husk at den innebygde `map` tar en n-arguments prosedyre og n liste-argumenter.

Eksempel på bruk av `map` + `reduce`

- Gitt to vektorer
  $a = \langle 2, 1, 3 \rangle$
  $b = \langle 1, 3, 0 \rangle$
- så beregnes prikk-produktet som
  $x \cdot y = \sum_i x_i \times y_i$
- $a \cdot b = 2 \times 1 + 1 \times 3 + 3 \times 0 = 5$
- Versjon 2 med modulær design: kjede av høyereordens operasjoner over sekvenser.
- I SICP-terminologi: sekvenser som *standardisert grensesnitt*.

```
(define a '(2 1 3))

(define b '(1 3 0))

(define (dot-product x y)
  (if (null? x)
      0
      (+ (* (car x) (car y))
         (dot-product
          (cdr x) (cdr y)))))

(dot-product a b) → 5

(define (dot-product x y)
  (reduce + 0 (map * x y)))
```

Fjerning av elementer

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


? (filter-odd '(1 2 3 4 5 6 7)) → (1 3 5 7)

? (filter-odd '()) → ()
```

- `filter-odd` returnerer en liste der alle partall er fjernet.
- Basistilfellet gir `'()`.
- Tester det første elementet i lista.
- `cons` samler opp resultatet dersom elementet tester sant.
- Rekurserer på `cdr` av lista.

Muligheter for generalisering igjen

- Igjen kan vi forestille oss et mer generelt mønster:

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

- Og igjen kan vi abstrahere mønsteret til en høyereordens prosedyre:
- Vi lar predikatet `pred` bli et parameter.

filter

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

? (filter odd?  '(0 1 2 3 4 5 6 7 8 9))
→ (1 3 5 7 9)

? (filter even?  '(0 1 2 3 4 5 6 7 8 9))
→ (0 2 4 6 8)

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

- Anvender et predikat på hvert element.
- Returnerer en ny liste med kun de elementene som predikatet tester sant for.
- (Hva slags type prosess gir `filter` opphav til?)

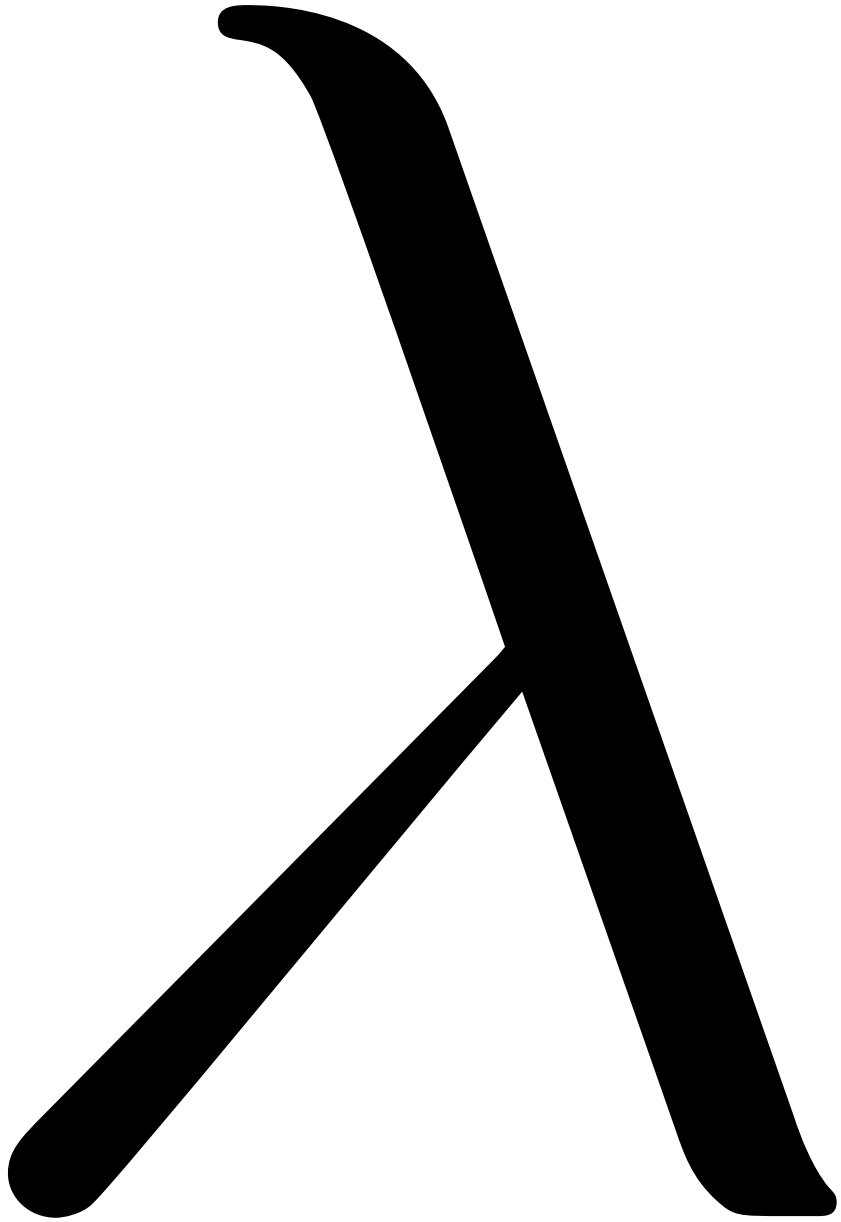
Halerekursjon og lister

```
(define (copy-list items)
  (if (null? items)
      '()
      (cons (car items)
            (copy-list (cdr items)))))

? (copy-list '(1 2 3 4)) → (1 2 3 4)

(define (copy-list items)
  (define (copy-iter in out)
    (if (null? in)
        out
        (copy-iter (cdr in)
                   (cons (car in) out))))
  (copy-iter items '()))

? (copy-list '(1 2 3 4)) → (4 3 2 1)
```

- Hva med halerekursjon når vi bygger opp lister?
- La oss forsøke å skrive en halerekursiv versjon av `copy-list`!
- Vi legger til en akkumulatorvariabel.
- Hva er problemet her?
- Bygger opp lista bakvendt.
- (Ser på halerekursjon på lister igjen når vi introduserer destruktive operasjoner.)

Høyereordens prosedyrer del 2

- Vi har så langt sett på prosedyrer som argumenter.
- Men vi kan øke uttrykkskraften betydelig når vi også tar med prosedyrer som returverdi.
- Enkelt: alt vi trenger å gjøre er å returnere et `lambda`-uttrykk.



Prosedyrer som returverdi

```
? (define (make-interval-test x y)
    (lambda (z)
      (and (> z x)
           (< z y))))

? (make-interval-test 2 7)
→  #<procedure>

? ((make-interval-test 2 7) 5)
→ #t

? (filter (make-interval-test 2 7)
          '(0 1 2 3 4 5 6 7 8 9))
→ (3 4 5 6)
```

- `make-interval-test` lager et predikat som vil sjekke om dens argument er innenfor et gitt intervall.
- `lambda`-uttrykket i kroppen *anvendes* ikke:
- Prosedyren som uttrykket evaluerer til er i seg selv *returverdien*.
- Så kan vi senere bruke den returnerte prosedyren som vi vil.

Prosedyrer som returverdi (forts.)

- Enkel søk-og-erstatt, *versjon 1*:

```
? (define (make-replacer x y)
    (lambda (z)
      (if (= z x)
          y
          z)))


? (map (make-replacer 42 'towel)
       '(1 42 2 42 3))
→ (1 towel 2 towel 3)


? (map (make-replacer 42 '(0))
       '(1 42 2 42 3))
→ (1 (0) 2 (0) 3)
```

Prosedyrer inn, prosedyrer ut

- Enkel søk-og-erstatt, *versjon 2*:

```
? (define (make-replacer pred proc)
    (lambda (z)
      (if (pred z)
          (proc z)
          z)))


? (map (make-replacer
          (make-interval-test 0 5)
          (lambda (x) (* x 10)))
     '(1 42 2 42 3))
→ (10 42 20 42 30)


? (map (make-replacer
          odd?
          (lambda (x) (+ 1 x)))
       '(1 42 2 42 3))
→ (2 42 2 42 4)
```

Prosedyrer inn, prosedyrer ut (forts.)

```
? (define (compose proc1 proc2)
    (lambda (x)
      (proc1 (proc2 x))))

? (define foo
    (compose (make-interval-test 5 10)
             square))

? (foo 2) → #f

? (foo 3) → #t

? (foo 4) → #f
```

- Et siste eksempel: `compose`.
- Setter sammen to prosedyrer til én ny.
- Her lager vi et nytt predikat som for et tall x sjekker om $x^2 \in (5, 10)$

Førsteklasses prosedyrer

- Prosedyrer er "førsteklasses borgere", dvs. at de kan brukes på akkurat samme måte som andre verdier i språket:
- Kan sendes som argument til en høyereordens prosedyre.
- Kan være returverdi fra en høyereordens prosedyre.
- Kan være anonyme eller bindes til variabler (et prosedyrenavn er bare en vanlig variabel som tilfeldigvis er bundet til en prosedyre).
- Kan lagres i datastrukturer:

```
? (define stuff (list + "zoo" 42 'towel))

? stuff → (#<procedure:+> "zoo" 42 towel)

? ((car stuff) 1 2) →  3
```

Neste gang

- Mer om `lambda`-uttrykk:
- Lokale variabler og `let`-uttrykk.
- Dataabstraksjon med lister