

IN2040: Funksjonell Programmering

Strømmer og utsatt evaluering

Martin Steffen

Universitetet i Oslo

9. uke (18. 10. 2022)



- ▶ Mer om (prosedyre)navn, **bindinger**, og verditilordning
- ▶ Nok en ny abstrakt muterbar datatype basert på lister: **tabeller**
- ▶ **Memoisering**



- ▶ Repetisjon: Anonym rekursjon
- ▶ Repetisjon: Sekvensoperasjoner
- ▶ **Strømmer**
- ▶ Realisering av data ved behov
- ▶ Uendelige datastrukturer
- ▶ **Utsatt evaluering**





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

```
? (fac 5) → 120
```



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

```
? (fac 5) → 120
```

- ▶ Rekursjon: selv-referanse.
- ▶ Er det mulig å utføre rekursjon med kun anonyme prosedyrer?
- ▶ Kan vi implementere `fac` uten rekursive kall? (Se SICP-oppgave 4.21.)



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

```
? (fac 5) → 120
```

- ▶ Rekursjon: selv-referanse.
- ▶ Er det mulig å utføre rekursjon med kun anonyme prosedyrer?
- ▶ Kan vi implementere `fac` uten rekursive kall? (Se SICP-oppgave 4.21.)

```
?
(lambda (n)
  (if (= n 1)
      1
      (* n (fac (- n 1)))))
5
→ 120
```



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

```
? (fac 5) → 120
```

- ▶ Rekursjon: selv-referanse.
- ▶ Er det mulig å utføre rekursjon med kun anonyme prosedyrer?
- ▶ Kan vi implementere `fac` uten rekursive kall? (Se SICP-oppgave 4.21.)

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

5
→ 120
```



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

```
? (fac 5) → 120
```

- ▶ Rekursjon: selv-referanse.
- ▶ Er det mulig å utføre rekursjon med kun anonyme prosedyrer?
- ▶ Kan vi implementere `fac` uten rekursive kall? (Se SICP-oppgave 4.21.)

```
? ((lambda (proc n)
  (proc proc n))
  (lambda (fac n)
    (if (= n 1)
        1
        (* n (fac fac (- n 1))))))
  5)
→ 120
```




- ▶ Vi har tidligere jobbet mye med sekvensoperasjoner definert for lister.
- ▶ F.eks. `map`, `reduce` (accumulate), `filter`, m.fl.

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

```
? (reduce * 1 '(2 2 2)) → 8
```



- ▶ Vi har tidligere jobbet mye med sekvensoperasjoner definert for lister.
- ▶ F.eks. `map`, `reduce` (accumulate), `filter`, m.fl.

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

```
? (reduce * 1 '(2 2 2)) → 8
```

```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))
```

```
? (interval 0 5) → (0 1 2 3 4 5)
```



- ▶ Vi har tidligere jobbet mye med sekvensoperasjoner definert for lister.
- ▶ F.eks. `map`, `reduce` (accumulate), `filter`, m.fl.

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

```
? (reduce * 1 '(2 2 2)) → 8
```

```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))
```

```
? (interval 0 5) → (0 1 2 3 4 5)
```

```
? (reduce + 0 (interval 0 5)) → 15
```



```
(define (mystery low high)
  (define (recurse count sum)
    (cond ((> count high) sum)
          ((prime? count) ; anta at prime? er gitt
           (recurse (+ count 1) (+ count sum)))
          (else (recurse (+ count 1) sum))))
  (recurse low 0))
```

? (mystery 1 5) →



```
(define (sum-primes low high)
  (define (recurse count sum)
    (cond ((> count high) sum)
          ((prime? count) ; anta at prime? er gitt
           (recurse (+ count 1) (+ count sum)))
          (else (recurse (+ count 1) sum))))
  (recurse low 0))
```

```
? (sum-primes 1 5) → 10
```



```
(define (sum-primes low high)
  (define (recurse count sum)
    (cond ((> count high) sum)
          ((prime? count) ; anta at prime? er gitt
           (recurse (+ count 1) (+ count sum)))
          (else (recurse (+ count 1) sum))))
  (recurse low 0))
```

? (sum-primes 1 5) → 10

```
(define (sum-primes low high)
  (reduce + 0 (filter prime? (interval low high))))
```

- ▶ Å uttrykke beregninger som manipulasjon av sekvenser kan være elegant, konsist og modulært, sammenliknet med 'inkrementell' kode.
- ▶ Men finnes det relevante forskjeller i tids- eller plasskompleksitet?



- ▶ I visse tilfeller kan teknikken bli veldig (og unødvendig) lite effektiv:

```
? (car (cdr (filter prime? (interval 100 100000)))) → 103
```



- ▶ I visse tilfeller kan teknikken bli veldig (og unødvendig) lite effektiv:

```
? (car (cdr (filter prime? (interval 100 100000)))) → 103
```

- ▶ Det beregnes én lang sekvens, så én til, men bare 4 elementer 'brukes'.



- ▶ I visse tilfeller kan teknikken bli veldig (og unødvendig) lite effektiv:

```
? (car (cdr (filter prime? (interval 100 100000)))) → 103
```

- ▶ Det beregnes én lang sekvens, så én til, men bare 4 elementer 'brukes'.
- ▶ Med en mer tradisjonell programmeringsstil ville vi gjort beregningen **inkrementelt**, med de ulike operasjonene sammenvevd.
- ▶ (F.eks ved å iterativt teste en tellervariabel, og returnere så fort vi hadde kommet til det andre primtallet.)



- ▶ I visse tilfeller kan teknikken bli veldig (og unødvendig) lite effektiv:

```
? (car (cdr (filter prime? (interval 100 100000)))) → 103
```

- ▶ Det beregnes én lang sekvens, så én til, men bare 4 elementer 'brukes'.
- ▶ Med en mer tradisjonell programmeringsstil ville vi gjort beregningen **inkrementelt**, med de ulike operasjonene sammenvevd.
- ▶ (F.eks ved å iterativt teste en tellervariabel, og returnere så fort vi hadde kommet til det andre primtallet.)
- ▶ Kan vi beholde den elegante strukturen ved sekvensoperasjonene uten å miste effektiviteten ved inkrementelle beregninger?
- ▶ Løsningen: **strømmer**.



```
? (stream-car  
  (stream-cdr  
    (stream-filter prime? (stream-interval 100 100000))))  
→ 103
```



```
? (stream-car  
  (stream-cdr  
    (stream-filter prime? (stream-interval 100 100000))))  
→ 103
```

```
? (stream-car  
  (stream-cdr  
    (stream-filter prime? (stream-interval 100 100000))))  
→ 103
```

- ▶ Ideen: elementene genereres ikke før vi ber om å få se dem.
- ▶ En strøm er altså **en sekvens som konstrueres mens den brukes**, men lar oss late som om hele sekvensen allerede finnes.
- ▶ Først beskrevet av Peter Landin i 1965.
- ▶ **Utsatt evaluering** (*delayed evaluation*).





- ▶ En strøm skal ha en 'kontrakt' som ligner på vanlige lister:

```
(stream-car (cons-stream x y)) ≡ x  
(stream-cdr (cons-stream x y)) ≡ y
```

- ▶ Har også konstanten `the-empty-stream` og predikatet `stream-null?`.
- ▶ (For øyeblikket skal vi bare late som om disse var innebygget i Scheme.)
- ▶ På overflaten kan vi bruke strømmer som om de var lister...
- ▶ men `cdr`-verdien av en strøm **evalueres ikke** ved `cons-stream`, men først ved `stream-cdr`!

- ▶ `cons-stream` gir et 'løfte' om at `cdr`-verdien kan beregnes ved behov.
- ▶ Først når vi bruker `stream-cdr` lages elementet (løftet innfris).
- ▶ Etterspørselen styrer beregningstidspunkt (*computing on demand*).



- ▶ `cons-stream` gir et 'løfte' om at `cdr`-verdien kan beregnes ved behov.
- ▶ Først når vi bruker `stream-cdr` lages elementet (løftet innfris).
- ▶ Etterspørselen styrer beregningstidspunkt (*computing on demand*).



```
? (cons (+ 1 2)
        (+ 3 4))
```



- ▶ `cons-stream` gir et 'løfte' om at `cdr`-verdien kan beregnes ved behov.
- ▶ Først når vi bruker `stream-cdr` lages elementet (løftet innfris).
- ▶ Etterspørselen styrer beregningstidspunkt (*computing on demand*).



```
? (cons (+ 1 2)
        (+ 3 4))
→ (3 . 7)
```

- ▶ `cons-stream` gir et 'løfte' om at `cdr`-verdien kan beregnes ved behov.
- ▶ Først når vi bruker `stream-cdr` lages elementet (løftet innfris).
- ▶ Etterspørselen styrer beregningstidspunkt (*computing on demand*).



```
? (cons (+ 1 2)
        (+ 3 4))
```

```
→ (3 . 7)
```

```
? (cons-stream (+ 1 2)
               (+ 3 4))
```

```
→
```

- ▶ `cons-stream` gir et 'løfte' om at `cdr`-verdien kan beregnes ved behov.
- ▶ Først når vi bruker `stream-cdr` lages elementet (løftet innfris).
- ▶ Etterspørselen styrer beregningstidspunkt (*computing on demand*).



```
? (cons (+ 1 2)
        (+ 3 4))
```

```
→ (3 . 7)
```

```
? (cons-stream (+ 1 2)
               (+ 3 4))
```

```
→ (3 . #<promise>)
```

- ▶ `cons-stream` gir et 'løfte' om at `cdr`-verdien kan beregnes ved behov.
- ▶ Først når vi bruker `stream-cdr` lages elementet (løftet innfris).
- ▶ Etterspørselen styrer beregningstidspunkt (*computing on demand*).



```
? (cons (+ 1 2)
        (+ 3 4))
```

```
→ (3 . 7)
```

```
? (cons-stream (+ 1 2)
               (+ 3 4))
```

```
→ (3 . #<promise>)
```

```
? (stream-cdr
   (cons-stream (+ 1 2)
                (+ 3 4)))
```

```
→
```

- ▶ `cons-stream` gir et 'løfte' om at `cdr`-verdien kan beregnes ved behov.
- ▶ Først når vi bruker `stream-cdr` lages elementet (løftet innfris).
- ▶ Etterspørselen styrer beregningstidspunkt (*computing on demand*).



```
? (cons (+ 1 2)
        (+ 3 4))
```

```
→ (3 . 7)
```

```
? (cons-stream (+ 1 2)
               (+ 3 4))
```

```
→ (3 . #<promise>)
```

```
? (stream-cdr
   (cons-stream (+ 1 2)
                (+ 3 4)))
```

```
→ 7
```



- ▶ Liknende grensesnitt som lister, så listeoperasjonene kan lett tilpasses:

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



- ▶ Liknende grensesnitt som lister, så listeoperasjonene kan lett tilpasses:

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



- ▶ Liknende grensesnitt som lister, så listeoperasjonene kan lett tilpasses:

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

```
(define (stream-filter pred seq)
  (cond ((stream-null? seq) the-empty-stream)
        ((pred (stream-car seq))
         (cons-stream
          (stream-car seq)
          (stream-filter pred (stream-cdr seq))))
        (else (stream-filter pred (stream-cdr seq)))))
```




```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))
```



```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))

(define (stream-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (stream-interval (+ low 1) high))))
```



```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))

(define (stream-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (stream-interval (+ low 1) high))))
```

- ▶ En strøm utsetter evalueringen av sin `cdr` inntil den blir etterspurt.



```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))

(define (stream-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (stream-interval (+ low 1) high))))
```

- En strøm utsetter evalueringen av sin `cdr` inntil den blir etterspurt.

```
? (interval 1 10) → (1 2 3 4 5 6 7 8 9 10)
```



```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))

(define (stream-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (stream-interval (+ low 1) high))))
```

- En strøm utsetter evalueringen av sin `cdr` inntil den blir etterspurt.

```
? (interval 1 10) → (1 2 3 4 5 6 7 8 9 10)
```

```
? (stream-interval 1 10) →
```



```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))

(define (stream-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (stream-interval (+ low 1) high))))
```

- En strøm utsetter evalueringen av sin `cdr` inntil den blir etterspurt.

```
? (interval 1 10) → (1 2 3 4 5 6 7 8 9 10)
? (stream-interval 1 10) → (1 . #<promise>)
```



```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))

(define (stream-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (stream-interval (+ low 1) high))))
```

- En strøm utsetter evalueringen av sin `cdr` inntil den blir etterspurt.

```
? (interval 1 10) → (1 2 3 4 5 6 7 8 9 10)
? (stream-interval 1 10) → (1 . #<promise>)
? (stream-cdr (stream-interval 1 10)) →
```

```
(define (interval low high)
  (if (> low high)
      '()
      (cons low (interval (+ low 1) high))))

(define (stream-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (stream-interval (+ low 1) high))))
```

- En strøm utsetter evalueringen av sin `cdr` inntil den blir etterspurt.

```
? (interval 1 10) → (1 2 3 4 5 6 7 8 9 10)
? (stream-interval 1 10) → (1 . #<promise>)
? (stream-cdr (stream-interval 1 10)) → (2 . #<promise>)
```




```
? (stream-car  
  (stream-cdr  
    (stream-filter prime? (stream-interval 100 100000))))
```

→ 103

```
? (car (cdr (filter prime? (interval 100 100000))))
```

→ 103

- ▶ Begge uttrykkene returnerer det andre primtallet større enn 100.
- ▶ Listeversjonen kaller predikatet ca 100.000 ganger og lager ca 110.000 cons-celler.
- ▶ Strømversjonen kaller predikatet 4 ganger og lager 6 cons celler.



Applicative-order evaluation

- ▶ **Evaluer argumentene:** kall prosedyre på **verdiene**.
- ▶ Standard i Scheme.
- ▶ Andre navn: *strict / eager evaluation, call-by-value*.



Applicative-order evaluation

- ▶ **Evaluer argumentene:** kall prosedyre på **verdiene**.
- ▶ Standard i Scheme.
- ▶ Andre navn: *strict / eager evaluation, call-by-value*.

Normal-order evaluation

- ▶ Prosedyren kalles med **argumentuttrykkene**: evalueres først ved behov.
- ▶ Andre navn: *non-strict / lazy evaluation, call-by-name, call-by-need*.

- ▶ Gir samme resultat så lenge vi holder oss til ren funksjonell kode (og endelige datastrukturer), men kan gi forskjeller i effektivitet.



Applicative-order evaluation

- ▶ **Evaluer argumentene:** kall prosedyre på **verdiene**.
- ▶ Standard i Scheme.
- ▶ Andre navn: *strict / eager evaluation, call-by-value*.

Normal-order evaluation

- ▶ Prosedyren kalles med **argumentuttrykkene**: evalueres først ved behov.
- ▶ Andre navn: *non-strict / lazy evaluation, call-by-name, call-by-need*.
- ▶ Gir samme resultat så lenge vi holder oss til ren funksjonell kode (og endelige datastrukturer), men kan gi forskjeller i effektivitet.
- ▶ **delay** gir oss kontroll til å velge selv når uttrykk evalueres.
- ▶ Skal se at den kan brukes for å implementere **cons-stream**.



- ▶ `delay` er en *special form* som tar et uttrykk som argument og returnerer et 'løfte' om at uttrykket kan evalueres senere.
- ▶ Prosedyren `force` lar oss innfri løftet: evaluerer uttrykk som har blitt 'satt på vent' med `delay`.

```
? (define foo (* 21 2))
```

```
? foo →
```



- ▶ `delay` er en *special form* som tar et uttrykk som argument og returnerer et 'løfte' om at uttrykket kan evalueres senere.
- ▶ Prosedyren `force` lar oss innfri løftet: evaluerer uttrykk som har blitt 'satt på vent' med `delay`.

```
? (define foo (* 21 2))
```

```
? foo → 42
```



- ▶ `delay` er en *special form* som tar et uttrykk som argument og returnerer et 'løfte' om at uttrykket kan evalueres senere.
- ▶ Prosedyren `force` lar oss innfri løftet: evaluerer uttrykk som har blitt 'satt på vent' med `delay`.

```
? (define foo (* 21 2))
```

```
? foo → 42
```

```
? (define bar (delay (* 21 2)))
```

```
? bar →
```



- ▶ `delay` er en *special form* som tar et uttrykk som argument og returnerer et 'løfte' om at uttrykket kan evalueres senere.
- ▶ Prosedyren `force` lar oss innfri løftet: evaluerer uttrykk som har blitt 'satt på vent' med `delay`.

```
? (define foo (* 21 2))  
?  
? foo → 42  
?  
? (define bar (delay (* 21 2)))  
?  
? bar → #<promise>
```




- ▶ `delay` er en *special form* som tar et uttrykk som argument og returnerer et 'løfte' om at uttrykket kan evalueres senere.
- ▶ Prosedyren `force` lar oss innfri løftet: evaluerer uttrykk som har blitt 'satt på vent' med `delay`.

```
? (define foo (* 21 2))  
?  
? foo → 42  
?  
? (define bar (delay (* 21 2)))  
?  
? bar → #<promise>  
?  
? (force bar) → 42
```

Å implementere strømmer i Scheme



- ▶ `cons-stream` en *special form* som bruker `delay` på `cdr`-argumentet:

```
(cons-stream x y) ≡ (cons x (delay y))
```



- ▶ `cons-stream` en *special form* som bruker `delay` på `cdr`-argumentet:

```
(cons-stream x y) ≡ (cons x (delay y))
```

- ▶ Hvorfor må `cons-stream` være en *special form* (ikke vanlig prosedyre)?



- ▶ `cons-stream` en *special form* som bruker `delay` på `cdr`-argumentet:

```
(cons-stream x y) ≡ (cons x (delay y))
```

- ▶ Hvorfor må `cons-stream` være en *special form* (ikke vanlig prosedyre)?
- ▶ Resten av strømgrensesnittet kan realiseres som vanlige prosedyrer:

```
(define (stream-car stream)  
  (car stream))
```



- ▶ `cons-stream` en *special form* som bruker `delay` på `cdr`-argumentet:

```
(cons-stream x y) ≡ (cons x (delay y))
```

- ▶ Hvorfor må `cons-stream` være en *special form* (ikke vanlig prosedyre)?
- ▶ Resten av strømgrensesnittet kan realiseres som vanlige prosedyrer:

```
(define (stream-car stream)
  (car stream))

(define (stream-cdr stream)
  ...)
```



- ▶ `cons-stream` en *special form* som bruker `delay` på `cdr`-argumentet:

```
(cons-stream x y) ≡ (cons x (delay y))
```

- ▶ Hvorfor må `cons-stream` være en *special form* (ikke vanlig prosedyre)?
- ▶ Resten av strømgrensesnittet kan realiseres som vanlige prosedyrer:

```
(define (stream-car stream)  
  (car stream))
```

```
(define (stream-cdr stream)  
  (force (cdr stream)))
```



- ▶ `cons-stream` en *special form* som bruker `delay` på `cdr`-argumentet:

```
(cons-stream x y) ≡ (cons x (delay y))
```

- ▶ Hvorfor må `cons-stream` være en *special form* (ikke vanlig prosedyre)?
- ▶ Resten av strømgrensesnittet kan realiseres som vanlige prosedyrer:

```
(define (stream-car stream)
  (car stream))

(define (stream-cdr stream)
  (force (cdr stream)))

(define the-empty-stream '())
```



- ▶ `cons-stream` en *special form* som bruker `delay` på `cdr`-argumentet:

```
(cons-stream x y) ≡ (cons x (delay y))
```

- ▶ Hvorfor må `cons-stream` være en *special form* (ikke vanlig prosedyre)?
- ▶ Resten av strømgrensesnittet kan realiseres som vanlige prosedyrer:

```
(define (stream-car stream)
  (car stream))

(define (stream-cdr stream)
  (force (cdr stream)))

(define the-empty-stream '())

(define (stream-null? stream)
  ...)
```




- ▶ `cons-stream` en *special form* som bruker `delay` på `cdr`-argumentet:

```
(cons-stream x y) ≡ (cons x (delay y))
```

- ▶ Hvorfor må `cons-stream` være en *special form* (ikke vanlig prosedyre)?
- ▶ Resten av strømgrensesnittet kan realiseres som vanlige prosedyrer:

```
(define (stream-car stream)
  (car stream))

(define (stream-cdr stream)
  (force (cdr stream)))

(define the-empty-stream '())

(define (stream-null? stream)
  (null? stream))
```



- ▶ delay og force er innebygd i Scheme, men det kan være opplysende å reflektere rundt hvordan vi kunne definert dem selv.

Hvordan så implementere delay og force?



- ▶ delay og force er innebygd i Scheme, men det kan være opplysende å reflektere rundt hvordan vi kunne definert dem selv.
- ▶ Evaluering av et uttrykk kan utsettes ved å gjøre det til prosedyrekropp:

```
(delay exp) ≡ (lambda () exp)
```

- ▶ Dette kan så regnes som et løfte om å beregne exp senere.

Hvordan så implementere delay og force?



- ▶ delay og force er **innebygd** i Scheme, men det kan være opplysende å reflektere rundt hvordan vi kunne definert dem selv.
- ▶ Evaluering av et uttrykk kan utsettes ved å gjøre det til prosedyrekropp:

```
(delay exp) ≡ (lambda () exp)
```

- ▶ Dette kan så regnes som et løfte om å beregne exp senere.
- ▶ For å utføre selve beregningen kaller vi bare prosedyreobjektet:

```
(define (force promise)
```

```
  ...
```

Hvordan så implementere delay og force?



- ▶ delay og force er **innebygd** i Scheme, men det kan være opplysende å reflektere rundt hvordan vi kunne definert dem selv.
- ▶ Evaluering av et uttrykk kan utsettes ved å gjøre det til prosedyrekropp:

```
(delay exp) ≡ (lambda () exp)
```

- ▶ Dette kan så regnes som et løfte om å beregne exp senere.
- ▶ For å utføre selve beregningen kaller vi bare prosedyreobjektet:

```
(define (force promise)  
  (promise))
```



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

Et eksempel i mer detalj



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)
```



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))
```




```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))
```



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))  
⇒ (cons 1 (lambda () (stream-interval 2 10)))
```



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))  
⇒ (cons 1 (lambda () (stream-interval 2 10)))  
⇒ (1 . (lambda () (stream-interval 2 10)))
```

Et eksempel i mer detalj



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))  
⇒ (cons 1 (lambda () (stream-interval 2 10)))  
⇒ (1 . (lambda () (stream-interval 2 10)))
```

```
? (stream-cdr s)
```

Et eksempel i mer detalj



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))  
⇒ (cons 1 (lambda () (stream-interval 2 10)))  
⇒ (1 . (lambda () (stream-interval 2 10)))
```

```
? (stream-cdr s)  
⇒ (stream-cdr (1 . (lambda () (stream-interval 2 10))))
```



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))  
⇒ (cons 1 (lambda () (stream-interval 2 10)))  
⇒ (1 . (lambda () (stream-interval 2 10)))
```

```
? (stream-cdr s)  
⇒ (stream-cdr (1 . (lambda () (stream-interval 2 10))))  
⇒ (force (cdr (1 . (lambda () (stream-interval 2 10)))))
```

Et eksempel i mer detalj



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))  
⇒ (cons 1 (lambda () (stream-interval 2 10)))  
⇒ (1 . (lambda () (stream-interval 2 10)))
```

```
? (stream-cdr s)  
⇒ (stream-cdr (1 . (lambda () (stream-interval 2 10))))  
⇒ (force (cdr (1 . (lambda () (stream-interval 2 10)))))  
⇒ (force (lambda () (stream-interval 2 10)))
```

Et eksempel i mer detalj



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))  
⇒ (cons 1 (lambda () (stream-interval 2 10)))  
⇒ (1 . (lambda () (stream-interval 2 10)))
```

```
? (stream-cdr s)  
⇒ (stream-cdr (1 . (lambda () (stream-interval 2 10))))  
⇒ (force (cdr (1 . (lambda () (stream-interval 2 10)))))  
⇒ (force (lambda () (stream-interval 2 10)))  
⇒ ((lambda () (stream-interval 2 10)))
```


Et eksempel i mer detalj



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))  
⇒ (cons 1 (lambda () (stream-interval 2 10)))  
⇒ (1 . (lambda () (stream-interval 2 10)))
```

```
? (stream-cdr s)  
⇒ (stream-cdr (1 . (lambda () (stream-interval 2 10))))  
⇒ (force (cdr (1 . (lambda () (stream-interval 2 10)))))  
⇒ (force (lambda () (stream-interval 2 10)))  
⇒ ((lambda () (stream-interval 2 10)))  
⇒ (stream-interval 2 10)
```

Et eksempel i mer detalj



```
? (define s (stream-interval 1 10))  
? s → (1 . #<promise>)  
? (stream-cdr s) → (2 . #<promise>)
```

- ▶ Hva skjer egentlig i kallene på `stream-interval` og `stream-cdr`?

```
? (stream-interval 1 10)  
⇒ (cons-stream 1 (stream-interval 2 10))  
⇒ (cons 1 (delay (stream-interval 2 10)))  
⇒ (cons 1 (lambda () (stream-interval 2 10)))  
⇒ (1 . (lambda () (stream-interval 2 10)))
```

```
? (stream-cdr s)  
⇒ (stream-cdr (1 . (lambda () (stream-interval 2 10))))  
⇒ (force (cdr (1 . (lambda () (stream-interval 2 10)))))  
⇒ (force (lambda () (stream-interval 2 10)))  
⇒ ((lambda () (stream-interval 2 10)))  
⇒ (stream-interval 2 10)  
⇒ (cons-stream 2 (stream-interval 3 10))  
...
```



```
(delay exp) ≡ (lambda () exp)
```

- ▶ Definisjonen av delay over er tilstrekkelig for utsatt evaluering...
- ▶ men den kan gjøres mye mer effektiv!
- ▶ I motsetning til i/o-‘strømmer’ støtter strømmer *random access*.
- ▶ Strømelementer kan brukes flere ganger:
- ▶ Med definisjonen så langt må vi da innfri samme løfte flere ganger.
- ▶ **Memoisering** kan bygges inn i delay.



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                  (set! forced? #t)))
          result)))
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
          result)))
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                  (set! forced? #t)))
          result)))
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
          result)))
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
          result)))
```




- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
          result)))
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
          result)))
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
          result)))
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
          result)))
```

```
? (define (foo x)
   (display "I was called!")
   x)
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)  
  (let ((forced? #f)  
        (result #f))  
    (lambda ()  
      (if (not forced?)  
          (begin (set! result (proc))  
                  (set! forced? #t)))  
          result)))
```

```
? (define (foo x)  
  (display "I was called!")  
  x)  
  
? (define bar (delay (foo 42)))
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)  
  (let ((forced? #f)  
        (result #f))  
    (lambda ()  
      (if (not forced?)  
          (begin (set! result (proc))  
                  (set! forced? #t)))  
          result)))
```

```
? (define (foo x)  
  (display "I was called!")  
  x)  
  
? (define bar (delay (foo 42)))  
  
? bar → #<promise>
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)  
  (let ((forced? #f)  
        (result #f))  
    (lambda ()  
      (if (not forced?)  
          (begin (set! result (proc))  
                  (set! forced? #t)))  
          result)))
```

```
? (define (foo x)  
  (display "I was called!")  
  x)  
  
? (define bar (delay (foo 42)))  
  
? bar → #<promise>  
  
? (force bar)
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)  
  (let ((forced? #f)  
        (result #f))  
    (lambda ()  
      (if (not forced?)  
          (begin (set! result (proc))  
                  (set! forced? #t)))  
          result)))
```

```
? (define (foo x)  
  (display "I was called!")  
  x)  
  
? (define bar (delay (foo 42)))  
? bar → #<promise>  
? (force bar)  
  
~> I was called!  
→ 42
```




- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
          result)))
```

```
? (define (foo x)
    (display "I was called!")
    x)
? (define bar (delay (foo 42)))
? bar → #<promise>
? (force bar)
~> I was called!
→ 42
? bar → #<promise>
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                 (set! forced? #t)))
          result)))
```

```
? (define (foo x)
      (display "I was called!")
      x)
? (define bar (delay (foo 42)))
? bar → #<promise>
? (force bar)
~> I was called!
→ 42
? bar → #<promise>
? (force bar) →
```



- Spesialisert memoisering: fungerer kun for prosedyrer uten argumenter.

```
(delay exp) ≡ (memoize (lambda () exp))
```

```
(define (memoize proc)
  (let ((forced? #f)
        (result #f))
    (lambda ()
      (if (not forced?)
          (begin (set! result (proc))
                  (set! forced? #t)))
          result)))
```

```
? (define (foo x)
  (display "I was called!")
  x)
? (define bar (delay (foo 42)))
? bar → #<promise>
? (force bar)
~> I was called!
→ 42
? bar → #<promise>
? (force bar) → 42
```



- ▶ Utsatt evaluering går ikke så bra i hop med **mutasjoner**:

```
? (define a 4)
? (define b (delay (+ 1 a)))
? (set! a 0)
? (force b) →
```



- ▶ Utsatt evaluering går ikke så bra i hop med **mutasjoner**:

```
? (define a 4)
```

```
? (define b (delay (+ 1 a)))
```

```
? (set! a 0)
```

```
? (force b) → 1
```

- ▶ Verdien til den utsatte evalueringen avhenger av tidspunktet for når den faktisk kalles: kan lett skape mye forvirring (og bugs)!



- ▶ Utsatt evaluering går ikke så bra i hop med **mutasjoner**:

```
? (define a 4)
? (define b (delay (+ 1 a)))
? (set! a 0)
? (force b) → 1
```

- ▶ Verdien til den utsatte evalueringen avhenger av tidspunktet for når den faktisk kalles: kan lett skape mye forvirring (og bugs)!
- ▶ Enda mer forvirrende hvis det er optimisert med memoisering i tillegg:

```
? (set! a 100)
? (force b) →
```



- ▶ Utsatt evaluering går ikke så bra i hop med **mutasjoner**:

```
? (define a 4)
? (define b (delay (+ 1 a)))
? (set! a 0)
? (force b) → 1
```

- ▶ Verdien til den utsatte evalueringen avhenger av tidspunktet for når den faktisk kalles: kan lett skape mye forvirring (og bugs)!
- ▶ Enda mer forvirrende hvis det er optimisert med memoisering i tillegg:

```
? (set! a 100)
? (force b) → 1
```



```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))

? (stream-ref (stream-interval 1 1000000) 7)
→ 8
```




```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))
```

```
? (stream-ref (stream-interval 1 1000000) 7)
```

```
→ 8
```

```
(define (show-stream stream n)
  (cond ((= n 0) (display "...\\n"))
        ((stream-null? stream) (newline))
        (else (display (stream-car stream))
              (display " ")
              (show-stream (stream-cdr stream) (- n 1)))))
```

```
? (show-stream (stream-interval 1 1000000) 7)
```

```
→ 1 2 3 4 5 6 7 ...
```



```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))

? (stream-ref (stream-interval 1 1000000) 7)
→ 8
```

```
(define (show-stream stream n)
  (cond ((= n 0) (display "...\n"))
        ((stream-null? stream) (newline))
        (else (display (stream-car stream))
              (display " ")
              (show-stream (stream-cdr stream) (- n 1)))))

? (show-stream (stream-interval 1 1000000) 7)
→ 1 2 3 4 5 6 7 ...
```

- ▶ Antar også at vi har `stream-map` fra obligen.



- ▶ Vi har sett at en strøm er en sekvens som konstrueres mens den brukes.
- ▶ Kan dermed definere en *uendelig* sekvens, uten å faktisk beregne den.
- ▶ De naturlige tallene via en rekursiv strømgenerator:

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



- ▶ Vi har sett at en strøm er en sekvens som konstrueres mens den brukes.
- ▶ Kan dermed definere en *uendelig* sekvens, uten å faktisk beregne den.
- ▶ De naturlige tallene via en rekursiv strømgenerator:

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



- ▶ Vi har sett at en strøm er en sekvens som konstrueres mens den brukes.
- ▶ Kan dermed definere en *uendelig* sekvens, uten å faktisk beregne den.
- ▶ De naturlige tallene via en rekursiv strømgenerator:

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

- ▶ **Veldefinert p.g.a. utsatt evaluering.**
 - ▶ Hva mangler i forhold til rekursjonen vi har sett før?
 - ▶ Hva ville skjedd om vi brukte `cons`?



- ▶ Vi har sett at en strøm er en sekvens som konstrueres mens den brukes.
- ▶ Kan dermed definere en *uendelig* sekvens, uten å faktisk beregne den.
- ▶ De naturlige tallene via en rekursiv strømgenerator:

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

```
? (define nats (integers-starting-from 1))
```

- ▶ **Veldefinert p.g.a. utsatt evaluering.**
 - ▶ Hva mangler i forhold til rekursjonen vi har sett før?
 - ▶ Hva ville skjedd om vi brukte cons?
- ▶ Kan kombineres med andre strømprosedyrer og strømmer, f.eks.

```
? (define odds (stream-filter odd? nats))
```

```
? (show-stream odds 5)  $\rightsquigarrow$  1 3 5 7 9 ...
```



- ▶ Kan også definere uendelige strømmer direkte ved å la strømmen referere til seg selv rekursivt. (SICP: 'implisitt strømdefinisjon')
- ▶ Veldefinert p.g.a. utsatt evaluering.

```
? (define ones (cons-stream 1 ones))
```



- ▶ Kan også definere uendelige strømmer direkte ved å la strømmen referere til seg selv rekursivt. (SICP: 'implisitt strømdefinisjon')
- ▶ Veldefinert p.g.a. utsatt evaluering.

```
? (define ones (cons-stream 1 ones))
```

```
? (define ones (cons 1 ones))
```




- ▶ Kan også definere uendelige strømmer direkte ved å la strømmen referere til seg selv rekursivt. (SICP: 'implisitt strømdefinisjon')
- ▶ Veldefinert p.g.a. utsatt evaluering.

```
? (define ones (cons-stream 1 ones))
```

```
? (define ones (cons 1 ones))  $\rightsquigarrow$  ones: undefined; error.
```



- ▶ Kan også definere uendelige strømmer direkte ved å la strømmen referere til seg selv rekursivt. (SICP: 'implisitt strømdefinisjon')
- ▶ Veldefinert p.g.a. utsatt evaluering.

```
? (define ones (cons-stream 1 ones))
```

```
? (define ones (cons 1 ones))  $\rightsquigarrow$  ones: undefined; error.
```

```
? (define (add-streams stream1 stream2)  
  (stream-map + stream1 stream2))
```



- ▶ Kan også definere uendelige strømmer direkte ved å la strømmen referere til seg selv rekursivt. (SICP: 'implisitt strømdefinisjon')
- ▶ Veldefinert p.g.a. utsatt evaluering.

```
? (define ones (cons-stream 1 ones))  
  
? (define ones (cons 1 ones)) ~> ones: undefined; error.  
  
? (define (add-streams stream1 stream2)  
  (stream-map + stream1 stream2))  
  
? (define nats  
  (cons-stream 1 (add-streams ones nats)))
```



- ▶ Kan også definere uendelige strømmer direkte ved å la strømmen referere til seg selv rekursivt. (SICP: 'implisitt strømdefinisjon')
- ▶ **Veldefinert p.g.a. utsatt evaluering.**

```
? (define ones (cons-stream 1 ones))  
  
? (define ones (cons 1 ones)) ~> ones: undefined; error.  
  
? (define (add-streams stream1 stream2)  
  (stream-map + stream1 stream2))  
  
? (define nats  
  (cons-stream 1 (add-streams ones nats)))
```

- ▶ Definisjonen av `nats` her fungerer fordi akkurat nok av strømmen er konstruert til at vi kan bruke den rekursivt i `add-streams`:
 - ▶ 2. element = 1. element + én
 - ▶ 3. element = 2. element + én
 - ▶ o.s.v.



$$\text{fib}(n) = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{ellers} \end{cases}$$

- ▶ Illustrerte trerekursive prosesser med en prosedyre for å beregne tall i **Fibonacci-rekken**:
- ▶ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 . . .
- ▶ Bortsett fra de to første er hvert tall summen av de to foregående.



$$\text{fib}(n) = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{ellers} \end{cases}$$

- ▶ Illustrerte trerekursive prosesser med en prosedyre for å beregne tall i **Fibonacci-rekken**:
- ▶ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...
- ▶ Bortsett fra de to første er hvert tall summen av de to foregående.

```
? (define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```



- ▶ Illustrerte trerekursive prosesser med en prosedyre for å beregne tall i **Fibonacci-rekken**:
- ▶ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...
- ▶ Bortsett fra de to første er hvert tall summen av de to foregående.

$$\text{fib}(n) = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{ellers} \end{cases}$$

```
? (define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```

```
? (fib 5) → 5
```

```
? (fib 6) → 8
```

```
? (fib 7) → 13      ;; = 5 + 8
```

```
? (fib 8) → 21      ;; = 8 + 13
```

```
? (fib 9) → 34      ;; = 13 + 21
```



- ▶ Kan bruke samme 'triks' som for nats til å definere fibonaccifølgen som en uendelig strøm `fibs`.
- ▶ Sørger for at akkurat nok av `fibs` finnes til at den kan brukes rekursivt:

```
? (define fibs
```

```
? (show-stream fibs 20)
```

```
~> 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

```
? (stream-ref fibs 9) → 34
```




- ▶ Kan bruke samme 'triks' som for nats til å definere fibonaccifølgen som en uendelig strøm `fibs`.
- ▶ Sørger for at akkurat nok av `fibs` finnes til at den kan brukes rekursivt:

```
? (define fibs
  (cons-stream 0
```

```
? (show-stream fibs 20)
```

```
~> 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

```
? (stream-ref fibs 9) → 34
```



- ▶ Kan bruke samme 'triks' som for nats til å definere fibonaccifølgen som en uendelig strøm `fibs`.
- ▶ Sørger for at akkurat nok av `fibs` finnes til at den kan brukes rekursivt:

```
? (define fibs
  (cons-stream 0
    (cons-stream 1
```

```
? (show-stream fibs 20)
```

```
~> 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

```
? (stream-ref fibs 9) → 34
```



- ▶ Kan bruke samme 'triks' som for nats til å definere fibonaccifølgen som en uendelig strøm `fibs`.
- ▶ Sørger for at akkurat nok av `fibs` finnes til at den kan brukes rekursivt:

```
? (define fibs
  (cons-stream 0
    (cons-stream 1
      (add-stream fibs))))

? (show-stream fibs 20)
~> 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

? (stream-ref fibs 9) → 34
```



- ▶ Kan bruke samme 'triks' som for nats til å definere fibonaccifølgen som en uendelig strøm `fibs`.
- ▶ Sørger for at akkurat nok av `fibs` finnes til at den kan brukes rekursivt:

```
? (define fibs
  (cons-stream 0
    (cons-stream 1
      (add-stream fibs (stream-cdr fibs)))))
```

```
? (show-stream fibs 20)
```

```
~> 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 .
```

```
? (stream-ref fibs 9) → 34
```



- ▶ Kan bruke samme 'triks' som for nats til å definere fibonaccifølgen som en uendelig strøm `fibs`.
- ▶ Sørger for at akkurat nok av `fibs` finnes til at den kan brukes rekursivt:

```
? (define fibs
  (cons-stream 0
    (cons-stream 1
      (add-stream fibs (stream-cdr fibs)))))
```

```
? (show-stream fibs 20)
```

```
~> 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 .
```

```
? (stream-ref fibs 9) → 34
```

- ▶ Antall addisjoner for det n -te elementet i strømmen (SICP 3.57):
 - ▶ Dersom delay er implementert *med* memoisering: Lineær.
 - ▶ *Uten*: Eksponentiell.



```
? (define (powers-of n)
  (define (iter x)
    (cons-stream x (iter (* x n))))
  (iter n))

? (show-stream (powers-of 2) 5) ~> 2 4 8 16 32 ...
```



```
? (define (powers-of n)
  (define (iter x)
    (cons-stream x (iter (* x n))))
  (iter n))
```

```
? (show-stream (powers-of 2) 5)  $\rightsquigarrow$  2 4 8 16 32 ...
```

```
? (define power-table (stream-map powers-of nats))
```



```
? (define (powers-of n)
  (define (iter x)
    (cons-stream x (iter (* x n))))
  (iter n))
```

```
? (show-stream (powers-of 2) 5)  $\rightsquigarrow$  2 4 8 16 32 ...
```

```
? (define power-table (stream-map powers-of nats))
```

```
? (show-stream (stream-ref power-table 0) 5)  $\rightsquigarrow$  1 1 1 1 1 ...
```

```
? (show-stream (stream-ref power-table 1) 5)  $\rightsquigarrow$  2 4 8 16 32 ...
```

```
? (show-stream (stream-ref power-table 2) 5)  $\rightsquigarrow$  3 9 27 81 243 ...
```

```
? (show-stream (stream-ref power-table 3) 5)  $\rightsquigarrow$  4 16 64 256 1024
```

```
? (show-stream (stream-ref power-table 4) 5)  $\rightsquigarrow$  5 25 125 625 3125
```

- Kan tenke på `power-table` som en tabell med uendelig mange og uendelig lange rader, der cellene ikke lages før vi trenger dem.



- ▶ Oblig 3a med prekode.
- ▶ Hovedfokus: memoisering + strømmer.

3a



- ▶ Mer om (uendelige) **strømmer** og **utsatt evaluering**.
- ▶ Hvordan definere egne *special forms*?
- ▶ **Makroer**: lar oss transformere kode før den blir evaluert eller kompilert.
- ▶ Strømmer, tid og tilstand.