# IN2040: Funksjonell Programmering

## *En Scheme-evaluator i Scheme, del 2*

Martin Steffen

Universitetet i Oslo

Uke # 12 (8. Nov 2022)

# Tema

## Forrige uke

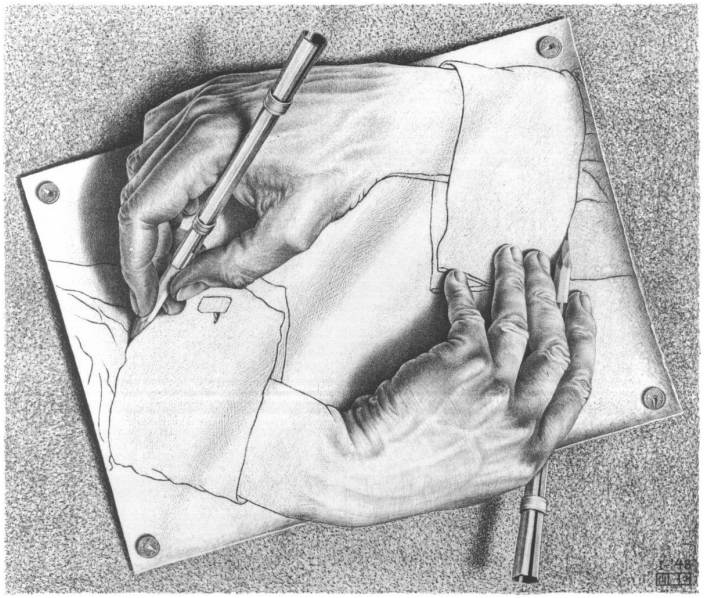
- SICP 4.1.
- Structure and *interpretation* of computer programs
- 'Metacircular evaluator'
- Scheme-evaluator skrevet i Scheme
- Omgivelsesmodellen


## I dag

- Vi skal se litt mer på koden fra 4.1 og alternativ syntaks.
- SICP 4.2 og alternativ semantikk: *lazy evaluation*.

# En evaluator for Scheme, i Scheme

- Evaluator (interpreter) –
  et program som tolker og utfører
  et program.
- Metasirkulær evaluator –
  en evaluator skrevet i samme
  språket som den skal evaluere.

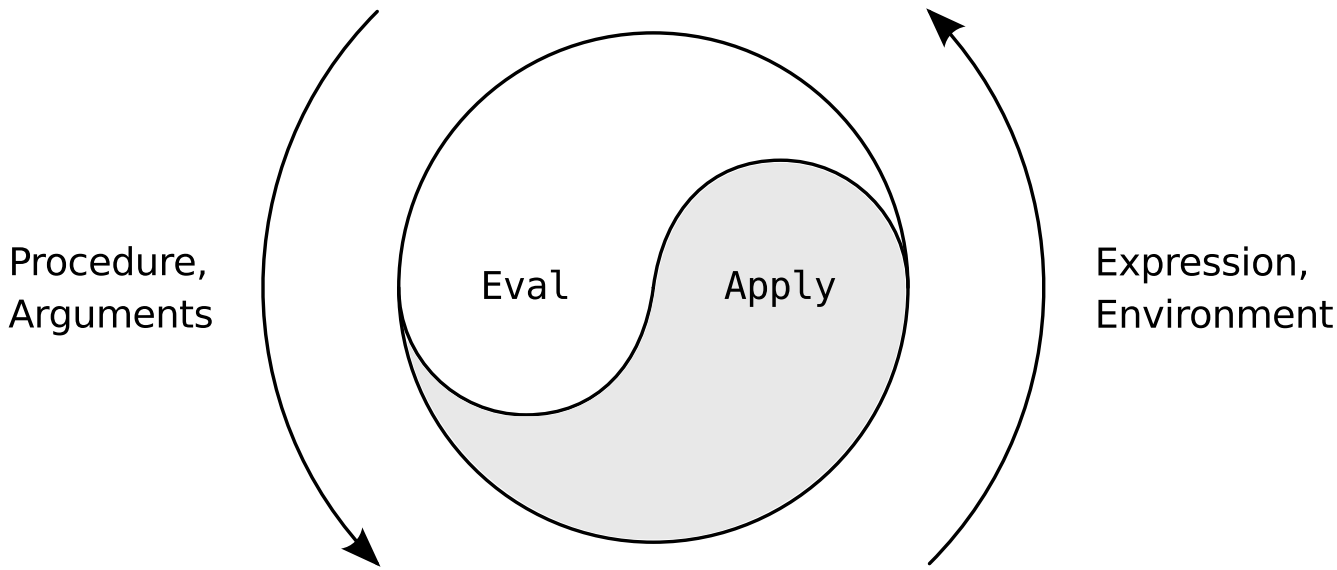


## Motivasjon

- Gjør omgivelsesmodellen eksplisitt gjennom en konkret implementasjon.
- Bedre forståelse for hvordan ulike uttrykk evalueres.
- Lar oss eksperimentere med alternativ syntaks og alternativ semantikk.
- Viser hvordan program er data.

# Delene i den metasirkulære evaluatoren

1. Datastrukturer for å representere omgivelser og prosedyreobjekter.
2. Predikater og aksessorer som definerer syntaksen til uttrykk i språket.
3. `mc-eval` og `mc-apply`
   - Med spesialiserte varianter for evaluering av *special forms*.
   - Definerer semantikken.
4. Mekanisme for å kalle primitive / innebygde prosedyrer.
5. REPL



# Program = data

- Evaluatoren vår er en prosedyre som tar kode som argument.
- Men for evaluatoren er koden bare symbolske data som vi kan manipulere på vanlig måte.

# Program = data

- Evaluatoren vår er en prosedyre som tar kode som argument.
- Men for evaluatoren er koden bare symbolske data som vi kan manipulere på vanlig måte.
- Merk at Scheme allerede gir oss tilgang til sin underliggende evaluator via den innebygde `eval` (som er grunnen til at vi kaller vår egen 'metacircular evaluator' for `mc-eval`).
- Den innebygde `apply` kjenner vi også fra før.

# Program = data

- Evaluatoren vår er en prosedyre som tar kode som argument.
- Men for evaluatoren er koden bare symbolske data som vi kan manipulere på vanlig måte.
- Merk at Scheme allerede gir oss tilgang til sin underliggende evaluator via den innebygde `eval` (som er grunnen til at vi kaller vår egen 'metacircular evaluator' for `mc-eval`).
- Den innebygde `apply` kjenner vi også fra før.

```
? (define expression (list 'if (list '= 0 1) "foo" (list '+ 1 2)))

? expression
→ (if (= 0 1) "foo" (+ 1 2))

? (eval expression (interaction-environment))
→ 3
```

# Seksjon 4.1.2: Representasjon av uttrykk

- Nøyaktig hvordan alle konstruktorene, aksessorene og predikatene for ulike uttrykk er definert er ikke så interessant i seg selv.
- Men hva disse abstraksjonene betyr er viktig.
- De definerer syntaksen for språket.
- Ved å endre på prosedyrene her kan vi endre på den syntaktiske representasjonen, f.eks:
- `(square := (lambda (x) (* x x)))`
- `(if` ⟨*test1*⟩ `then` ⟨*alt1*⟩ `elsif` ⟨*test2*⟩ `then` ⟨*alt2*⟩ `else` ⟨*alt3*⟩`)`

# Språkdesign

- Et høynivåspråk som Lisp lar oss lage en prototype-evaluator raskt.
- Å ha en evaluator kan være et viktig først skritt på veien i språkdesign.
- Enkelt å teste ideer for hvordan språket skal se ut og tolkes.
- Vi skal se på en variant av Scheme der argumentene til prosedyrer evalueres på en annen måte enn vi er vant til:
- Lazy Scheme.

# To typer evaluering

- Eager evaluation

(Sussman)

- Lazy evaluation

(Abelson)

# To typer evaluering

- Eager evaluation
- Eller *applicative-order* / *strict evaluation*.
- Argumentene evalueres før en prosedyre anvendes.

(Sussman)

- Lazy evaluation

(Abelson)

# To typer evaluering

- Eager evaluation
- Eller *applicative-order* / *strict evaluation*.
- Argumentene evalueres før en prosedyre anvendes.

(Sussman)

- Lazy evaluation
- Eller *normal-order* / *non-strict evaluation*.
- Evaluering av argumentene utsettes til vi faktisk trenger verdiene deres.

(Abelson)

# Evalueringsstrategier: *applicative* vs. *normal-order*

- *Applicative-order* (*call-by-value*) er standardstrategien i Scheme.
- *Normal-order* ved *special-forms*, som `if`, `and` og `or`.

# Evalueringsstrategier: *applicative* vs. *normal-order*

- *Applicative-order* (*call-by-value*) er standardstrategien i Scheme.
- *Normal-order* ved *special-forms*, som `if`, `and` og `or`.

```
? (define (proc-if test then else)
    (if test then else))

? (define (foo) (foo))
```

```
? (proc-if #f (foo) 'bar) →

? (if #f (foo) 'bar) →
```

# Evalueringsstrategier: *applicative* vs. *normal-order*

- *Applicative-order* (*call-by-value*) er standardstrategien i Scheme.
- *Normal-order* ved *special-forms*, som `if`, `and` og `or`.

```
? (define (proc-if test then else)
    (if test then else))

? (define (foo) (foo))
```

```
? (proc-if #f (foo) 'bar) → uendelig løkke...

? (if #f (foo) 'bar) → bar
```

# Evalueringsstrategier: *applicative* vs. *normal-order*

- *Applicative-order* (*call-by-value*) er standardstrategien i Scheme.
- *Normal-order* ved *special-forms*, som `if`, `and` og `or`.

```
? (define (proc-if test then else)
    (if test then else))

? (define (foo) (foo))
```

```
? (proc-if #f (foo) 'bar) → uendelig løkke...

? (if #f (foo) 'bar) → bar
```

```
? (proc-if #f (/ 1 0) 'bar) →

? (if #f (/ 1 0) 'bar) →
```

# Evalueringsstrategier: *applicative* vs. *normal-order*

- *Applicative-order* (*call-by-value*) er standardstrategien i Scheme.
- *Normal-order* ved *special-forms*, som `if`, `and` og `or`.

```
? (define (proc-if test then else)
    (if test then else))

? (define (foo) (foo))
```

```
? (proc-if #f (foo) 'bar) → uendelig løkke...

? (if #f (foo) 'bar) → bar
```

```
? (proc-if #f (/ 1 0) 'bar) → error; division by zero

? (if #f (/ 1 0) 'bar) → bar
```

# Evalueringsstrategier: *applicative* vs. *normal-order*

- *Applicative-order* (*call-by-value*) er standardstrategien i Scheme.
- *Normal-order* ved *special-forms*, som `if`, `and` og `or`.

```
? (define (proc-if test then else)
    (if test then else))

? (define (foo) (foo))
```

```
? (proc-if #f (foo) 'bar) → uendelig løkke...

? (if #f (foo) 'bar) → bar
```

```
? (proc-if #f (/ 1 0) 'bar) → error; division by zero

? (if #f (/ 1 0) 'bar) → bar
```

- En funksjon med *non-strict* semantikk kan ha en *veldefinert* verdi selv for argumenter som *ikke* er veldefinerte.
- Vi har også sett hvordan utsatt evaluering gjør det mulig å jobbe med uendelige datastrukturer, som strømmer.

# Lazy evaluation

- I noen varianter av *normal-order evaluation* re-evalueres uttrykkene hver gang de brukes (*call-by-name*).
- Men *lazy evaluation* inkluderer gjerne også memoisering:
  - Argument-uttrykk evalueres kun når de trengs og maksimalt én gang.
  - Ved evaluering lagres verdien for mulig gjenbruk senere.
  - *Call-by-need*

# Lazy evaluation

- I noen varianter av *normal-order evaluation* re-evalueres uttrykkene hver gang de brukes (*call-by-name*).
- Men *lazy evaluation* inkluderer gjerne også memoisering:
  - Argument-uttrykk evalueres kun når de trengs og maksimalt én gang.
  - Ved evaluering lagres verdien for mulig gjenbruk senere.
  - *Call-by-need*

- Vi skal gjøre Scheme-evaluatoren vår *lazy*.
  - Vi trenger altså å kunne utsette evaluering av uttrykk, som *thunks*:
  - I stedet for å evaluere argumentene før et prosedyrekall pakker vi sammen uttrykkene og kall-omgivelsen slik at de kan evalueres senere.
  - (Gjelder kun ikke-primitive prosedyrer.)

# Thunks

```
(define (delay-it exp env)
  (list 'thunk exp env))

(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk)
  (cadr thunk))

(define (thunk-env thunk)
  (caddr thunk))
```

- Representerer uttrykk som *thunks*.
  - Liste av taggen `'thunk`, selve uttrykket, og omgivelsen.

# Thunks

```
(define (delay-it exp env)
  (list 'thunk exp env))

(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk)
  (cadr thunk))

(define (thunk-env thunk)
  (caddr thunk))

(define (evaluated-thunk? obj)
  (tagged-list? obj
                'evaluated-thunk))

(define (thunk-value thunk)
  (cadr thunk))
```

- Representerer uttrykk som *thunks*.
  - Liste av taggen `'thunk`, selve uttrykket, og omgivelsen.
- Thunk som allerede har blitt evaluert (og verdien memoisert):
  - Taggen `'evaluated-thunk`
  - og verdien til uttrykket.
  - Trenger ikke huske omgivelsen.

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (force-it, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (force-it, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval                ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (force-it, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)     ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())        ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)            ; memoisert?
         (thunk-value obj))
        (else obj)))                       ; ikke en thunk.
```

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

- mangler det noen?

# Thunks (`force-it`, men ikke ferdig ennå)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (mc-eval              ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)    ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())       ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)           ; memoisert?
         (thunk-value obj))
        (else obj)))                      ; ikke en thunk.
```

- mangler det noen?
- hva hvis `mc-eval` gir en thunk?

# Thunks (force-it)

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value       ; evaluer uttrykket.
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)     ; erstatter uttrykk med verdi.
           (set-cdr! (cdr obj) '())        ; glem omgivelsen.
           result))
        ((evaluated-thunk? obj)            ; memoisert?
         (thunk-value obj))
        (else obj)))                       ; ikke en thunk.

(define (actual-value exp env)
  (force-it (mc-eval exp env)))

; gjensidig rekursjon: actual-value kaller force-it igjen
; i tilfelle mc-eval returnerer en ny thunk.
```

# Lazy Scheme

- Med thunks har vi nå en måte å 'fryse' uttrykk på.
- Så må dette plugges inn i `mc-eval` / `mc-apply`.
- Må opprette thunks for argumentuttrykk når vi skal evaluere prosedyrer.
- Generelt kan vi si at uttrykk kun trenger å evalueres dersom de angir
  - argumentene til primitive prosedyrer,
  - operatoren vi skal til å anvende i et prosedyrekall,
  - testen i en kontrollstruktur (dvs. `if`), eller
  - de skal printes til REPLet.
  - (Skal se på de to første tilfellene sammen.)

# Cf. “eager” mc-eval

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((special-form? exp) (eval-special-form exp env))
        ((application? exp)
         (mc-apply (mc-eval (operator exp) env)
                   (list-of-values (operands exp) env)))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (mc-eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

# mc-eval (“lazy”)

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((special-form? exp) (eval-special-form exp env))
        ((application? exp)
         (mc-apply (actual-value (operator exp) env)
                   (operands exp)
                   env))))
```

# mc-eval ("lazy")

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((special-form? exp) (eval-special-form exp env))
        ((application? exp)
         (mc-apply (actual-value (operator exp) env)
                   (operands exp)  ;; Selve uttrykkene, ikke verdiene.
                   env))))         ;; Trenger også kallomgivelsen.
```

# Cf. “eager” mc-apply (repetisjon)

```
(define (mc-apply proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         (eval-sequence (procedure-body proc)
                        (extend-environment
                         (procedure-parameters proc)
                         args
                         (procedure-environment proc))))))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (mc-eval (first-exp exps) env))
        (else (mc-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

- Klassifiserer prosedyrer som primitive eller sammensatte.
- Ved sammensatte prosedyrer: Evaluerer hvert uttrykk i prosedyre-kroppen i en ny omgivelse med de aktuelle parameter-bindingene.

# Cf. "eager" mc-apply (repetisjon)

```
(define (mc-apply proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         (eval-sequence (procedure-body proc)
                        (extend-environment
                         (procedure-parameters proc)
                         args
                         (procedure-environment proc))))))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (mc-eval (first-exp exps) env))
        (else (mc-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

- Klassifiserer prosedyrer som primitive eller sammensatte.
- Ved sammensatte prosedyrer: Evaluerer hvert uttrykk i prosedyre-kroppen i en ny omgivelse med de aktuelle parameter-bindingene.

# mc-apply

```
(define (mc-apply proc args env)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure
          proc (list-of-arg-values args env)))  ; Henter faktiske verdier
        ((compound-procedure? proc)
         (eval-sequence
          (procedure-body proc)
          (extend-environment
           (procedure-parameters proc)          ; Parameterne bindes
           (list-of-delayed-args args env)      ; til thunks.
           (procedure-environment proc))))))

(define (list-of-arg-values exps env)
  (if (no-operands? exps) '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps) env))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps) '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps) env))))
```

# Strømmer som late lister

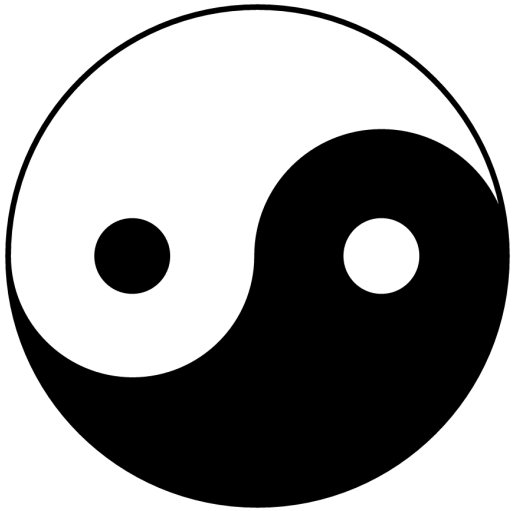
- Tidligere har vi implementert strømmer som 'utsatte' lister:
- Brukte spesialformene `delay` og `cons-stream`.
- Trenger ikke lenger skille mellom strømmer og lister:
- La `cons` bruke *lazy evaluation*.
- Liste-operasjoner vi definerer på vanlig måte vil dermed også fungere for uendelige sekvenser.
- Må i så fall implementere `cons` som en ikke-primitiv, f.eks. representert som prosedyrer slik vi har sett tidligere:

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

(define (car z)
  (z (lambda (p q) p)))

(define (cdr z)
  (z (lambda (p q) q)))
```

# NB

- Rekkefølgen på operasjoner viktig hvis vi tillater side-effekter.
- Tidspunktet for evaluering avgjørende for effekt.
- Kan være vanskelig å forutsi ved *lazy evaluation*.
- Memoiseringen gjør det enda vanskeligere (tidspunkt for førstegangs evaluering avgjørende).
- Et av få språk med *lazy evaluation* som standardstrategi er Haskell.
- Et mere rent funksjonelt språk.

# En slags veiledning i hvordan lese kap. 4



- Trenger ikke finlese all koden.
- Fokuser på prinsippene.
- Gjelder spesielt de syntaktiske abstraksjonene i seksjon 4.1.2.
- Spor evalueringen av en egendefinert prosedyre, og et par spesialformer.
- Med en metasirkulær evaluator blir det ekstra viktig å være var på skillet mellom *implementeringsspråk* og *implementert språk*.
- God øvelse: la meta-Scheme bruke norsk.
- Ikke fokuser på hvordan primitive / innebygde prosedyrer behandles.
- Merk hvordan omgivelsesmodellen og evaluatoren speiler hverandre.
- Merk samspillet mellom `eval` og `apply`.