

IN2040: Funksjonell Programmering

En Scheme-evaluator i Scheme

Martin Steffen

Universitetet i Oslo

11. uke (1. Nov. 2022)





Forrige forelesning

- ▶ Strømmer og utsatt evaluering
- ▶ Kort om makroer

I dag

- ▶ Kap. 4
- ▶ 'Metasirkulær evaluator'
- ▶ En interpreter / evaluator for Scheme skrevet i Scheme.

- ▶ I de neste to forelesningene går vi sammen igjennom *evaluator.scm* med kjøringseksempler.
- ▶ Prekoden til *oblig 3b*, se obligsiden.



- ▶ Vi har jobbet mye med **abstraksjon** i IN2040.
 - ▶ Vi har definert abstraksjonsbarrierer.
 - ▶ F.eks prosedyrer for å jobbe med mengder, bygget på prosedyrer for å jobbe med binærtrær.
 - ▶ Kan sees som dedikerte mini-språk i språket.
- ▶ Som programmerere er det viktig å tenke at vi ikke bare *braker* språk men *skaper* dem, lag på lag.
- ▶ Slik er det også med programmeringspråk; høynivåspråk bygger på lavnivåspråk som bygger på maskinspråk.
- ▶ Strategien med å løse et komplekst problem ved å lage oss et nytt tilpasset språk kalles (litt høytidelig) for **metalingvistisk abstraksjon**.



- ▶ Så langt: programmer for å utføre en gitt funksjon.
- ▶ Nå: programmer for å utføre programmer.
- ▶ **Evaluator:**
 - ▶ Et program som utfører instruksjonene skrevet i et programmeringsspråk.
- ▶ En evaluator for et program er altså bare nok et program.
 - ▶ En innsikt som kan virke selvsagt men likevel er fundamental og dyp!



- ▶ Så langt: programmer for å utføre en gitt funksjon.
- ▶ Nå: programmer for å utføre programmer.
- ▶ **Evaluator:**
 - ▶ Et program som utfører instruksjonene skrevet i et programmeringsspråk.
- ▶ En evaluator for et program er altså bare nok et program.
 - ▶ En innsikt som kan virke selvsagt men likevel er fundamental og dyp!
- ▶ I kap. 4 skal vi skrive en '**metasirkulær evaluator**' – en evaluator skrevet i samme språket som den skal evaluere:
- ▶ En evaluator for Scheme, i Scheme.

Hvorfor en evaluator for Scheme i Scheme?!



- ▶ Kap. 4 kan tidvis kanskje virke mystisk, men har egentlig til hensikt å **avmystifisere**.
- ▶ Gjør **omgivelsesmodellen** eksplisitt.
- ▶ **Bedre forståelse** for hvordan ulike uttrykk evalueres.
- ▶ Kan prøve ut **alternativ syntaks og semantikk**.
- ▶ Illustrerer en generell evaluator-struktur som også vil kunne brukes for andre språk.
- ▶ Viser tydelig hvordan **program er data**.





- ▶ En **omgivelse** er en sekvens av **rammer**.
- ▶ En ramme er en tabell som binder variabelnavn til verdier.
- ▶ Den **globale omgivelsen** består av én enkelt ramme (bl.a. med bindingene for primitive prosedyrer).
- ▶ **Variabler** har sin verdi relativ til en gitt omgivelse, definert ved den første rammen med en binding.
- ▶ **Bindinger** i en ramme opprettes via `define` eller som formelle parametre ved prosedyrekall.
- ▶ Bindinger kan endres med `set!`.
- ▶ En **prosedyre** lages ved å evaluere et `lambda`-uttrykk i en gitt omgivelse:
- ▶ Resultatet er et prosedyreobjekt som består av formelle parametere, kropp, og en peker til omgivelsen der den ble opprettet.

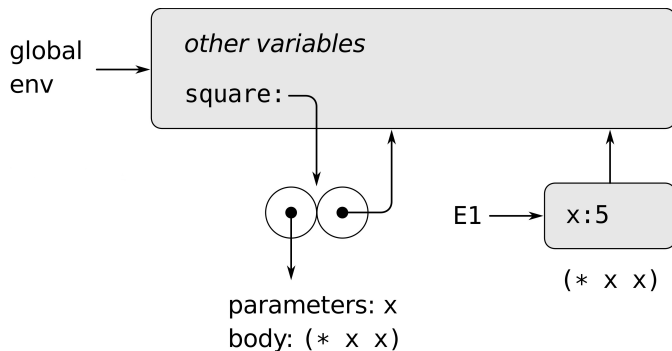
Litt repetisjon (forts.)



- ▶ Omgivelser, rammer, og prosedyrer.

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

```
? (square 5)
```



1. Abstrakte datastrukturer for å representere omgivelser, prosedyrer, etc.
2. Diverse prosedyrer som definerer syntaksen til uttrykkene i språket.
3. `eval` og `apply`
 - ▶ og spesialiserte varianter av disse som gir semantikken til *special forms*.
4. Mekanisme for å kalle primitive / innebygde prosedyrer.
5. REPL





- ▶ Evaluatoren implementerer en omgivelse som en liste av rammer.
 - ▶ En sekvens av tabeller med symbolbindinger.
 - ▶ `cdr` gir oss den omsluttende omgivelsen.
- ▶ Rammer er implementert som par av to lister;
 - ▶ en med symbolene og
 - ▶ en med de tilsvarende verdiene.

- ▶ Evaluatoren er en implementasjon av omgivelsesmodellen for evaluering.



Evalueringsreglene i omgivelsesmodellen

- ▶ For å **evaluere** et sammensatt uttrykk,
- ▶ evaluer del-uttrykkene rekursivt
- ▶ og **anvend** prosedyre-verdien på argument-verdiene.



Evalueringsreglene i omgivelsesmodellen

- ▶ For å **evaluere** et sammensatt uttrykk,
 - ▶ evaluer del-uttrykkene rekursivt
 - ▶ og **anvend** prosedyre-verdien på argument-verdiene.

- ▶ For å **anvende** en prosedyre på argumenter,
 - ▶ utvid omgivelsen til prosedyrobjectet med en ny ramme der de formelle parameterene er bundet til argumentene prosedyren kalles på,
 - ▶ og **evaluer** prosedyrekroppen i denne nye omgivelsen.



Evalueringsreglene i omgivelsesmodellen

- ▶ For å **evaluere** et sammensatt uttrykk,
 - ▶ evaluer del-uttrykkene rekursivt
 - ▶ og **anvend** prosedyre-verdien på argument-verdiene.

- ▶ For å **anvende** en prosedyre på argumenter,
 - ▶ utvid omgivelsen til prosedyrobjectet med en ny ramme der de formelle parameterene er bundet til argumentene prosedyren kalles på,
 - ▶ og **evaluer** prosedyrekroppen i denne nye omgivelsen.

- ▶ Fortsetter helt til uttrykkene er redusert til selv-evaluerende atomer og primitive prosedyrer (kalles direkte uten ny omgivelse).
- ▶ *Special forms* har sine egne evalueringsregler.



- ▶ Evaluatoren vår er en prosedyre som tar programkode som argument.
- ▶ Men for evaluatoren er koden bare symbolske data som vi kan manipulere på vanlig måte.
- ▶ Vi trenger prosedyrer for å klassifisere og plukke fra hverandre uttrykk.
 - ▶ Implementert som **abstraksjonsbarrierer**.
 - ▶ F.eks; i stedet for å hardkode alle steder at definisjoner er lister som begynner med symbolet `define` bruker vi et predikat `definition`?

```
(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))))
```

- ▶ Tar et uttrykk og en omgivelse: Analyserer typen til uttrykket (basert på tagged-list?) og evaluerer i henhold til dette.
- ▶ Hver *special form* krever sin egen behandling.
- ▶ Ved prosedyrekall: Evaluerer rekursivt operator- og operand-uttrykkene og kaller mc-apply på resultatet.



```
(define (eval-special-form exp env)
  (cond ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                          (lambda-body exp)
                          env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (mc-eval (cond->if exp) env))))
```

- ▶ Alle *special forms* har egne evalueringsregler. F.eks:
- ▶ *if*: evaluer konsekventen dersom predikatet evaluerer til sant, eller alternativet om det er usant.
- ▶ *lambda*: gjøres om til en liste av parametere, kropp og omgivelse, tagget med 'procedure.

```
(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))))))

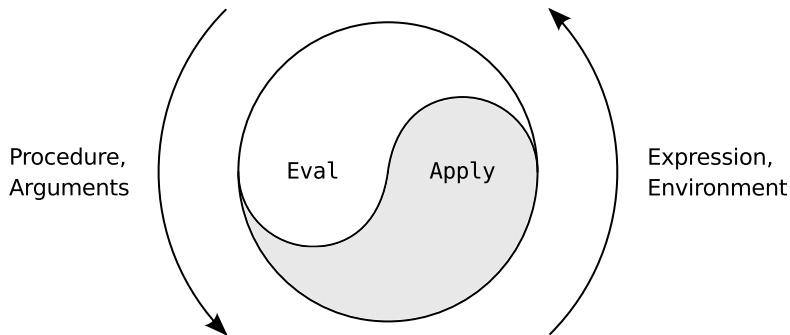
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (mc-eval (first-exp exps) env))
        (else (mc-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

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

```
(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 prosedyrekroppen i en ny omgivelse med de aktuelle parameter-bindingene.



Kjernen i evaluatoren

- ▶ **eval**; tar et uttrykk og en omgivelse, og kaller **apply** med en prosedyre og argumenter.
- ▶ **apply**; tar en prosedyre og argumenter, og kaller **eval** med et uttrykk og en omgivelse.
- ▶ Hvordan får vi noen gang et svar? Hva er basistilfellene?



```
(define (read-eval-print-loop) ;;tilsvarer driver-loop i SICP
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (mc-eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
    (read-eval-print-loop)))
```

- ▶ Vi kan bruke `read` siden vi evaluerer Scheme/Lisp.
 - ▶ Leser inn Scheme-uttrykk i én jafs.
- ▶ Hvis vi skulle skrevet en interpreter for et helt annet språk måtte vi heller bruke f.eks `read-char`.



- ▶ Mengden kode kan virke overveldende.
- ▶ Det viktige er ikke å ha finlest all koden men å forstå **prinsippene**.
- ▶ Spesielt gjelder dette de syntaktiske abstraksjonene i seksjon 4.1.2.
- ▶ Ta gjerne for deg én eller to *special forms*, f.eks `if` og `lambda` og sett deg inn i hvordan de fungerer.
- ▶ Ikke fokuser på hvordan primitive / innebygde prosedyrer behandles; dette er delvis magi og delvis en distraksjon fra de viktige ideene her!
- ▶ **Merk hvordan omgivelsesmodellen og evaluatoren speiler hverandre.**
- ▶ **Og samspillet mellom `eval` og `apply`.**
- ▶ Med en metasirkulær evaluator blir det ekstra viktig å være var på skillet mellom *implementeringsspråk* og *implementert språk*.

- ▶ Mer meta.
- ▶ Mer sirkularitet.
- ▶ Endringer av syntaksen og semantikken til språket:
- ▶ Implementering av *lazy Scheme*.

