

Innlevering 3b (INF2810, vår 2013)

- Dette er del to av den tredje obligatoriske oppgaven og dermed siste innlevering i INF2810. Man må ha minst 12 poeng tilsammen for 3a + 3b, og man kan få opptil 10 poeng på hver innlevering.
- I denne oppgaven skal vi jobbe med å implementere endringer i koden til den metasirkulære evaluatoren som beskrives i SICP seksjon 4.1.1–4.1.4. Forelesningsnotatene fra 26/4 vil også være nyttige her. Koden ligger ute på emnesiden: www.uio.no/studier/emner/matnat/ifi/INF2810/v13/undervisningsmateriale/evaluator.scm. Merk at koden i ‘*evaluator.scm*’ avviker litt her og der fra slik den ser ut i kap. 4.1, men dette er dokumentert med kommentarer.
- Lever en kopi av fila ‘*evaluator.scm*’ der du har gjort endringene som kreves av oppgavene. Det er viktig at du legger inn kommentarer som gjør det tydelig *hvor* i koden du har gjort endringer og for *hvilke* oppgaver. Der oppgavene også krever tekstlige svar eller kjøringseksempler legges dette inn som kommentarer.
- Oppgavene skal løses gruppevis (med mindre du har avtale om individuell levering). Som før holder det at én på hver gruppe leverer, men si i fra klart og tydelig i begynnelsen av besvarelsen hvem andre som er med på gruppa. Svar leveres via Devilry *innen utgangen av torsdag 16. mai*.

1 Bli kjent med evaluatoren

Last inn koden ‘*evaluator.scm*’. For å teste uttrykk mot den metasirkulære evaluatoren kan vi enten bruke dens egen (meta-)REPL – som startes ved å kalle (`read-eval-print-loop`) – eller vi kan sende uttrykk direkte til `mc-eval`. Men begge deler forutsetter at vi først initialiserer den globale omgivelsen ved å evaluere følgende

```
> (set! the-global-environment (setup-environment))
```

Da er alt klart for å evaluere uttrykk i den nye Scheme-implementasjonen vår. F.eks:

```
> (set! the-global-environment (setup-environment))
> (mc-eval '(+ 1 2) the-global-environment)
3
> (read-eval-print-loop)
```

```
;;; MC-Eval input:
(+ 1 2)
```

```
;;; MC-Eval value:
3
```

Tips: For å avbryte (meta-)read-eval-print-loopen kan dere f.eks trykke på EOF-symbolet til høyre på input-feltet. Prøv dere frem med noen forskjellige uttrykk, forsøk å definere noen prosedyrer, og bruk litt tid på å bli kjent med evaluatoren og orientere deg i koden. Merk at evaluatoren slik den står ikke har med støtte for alle innebygde primitiver i Scheme så det kan hende du må legge inn flere primitiver selv hvis det er noe du savner (se lista `primitive-procedures`).

- (a) Start REPL'en for den metasirkulære evaluatoren og evaluer følgende uttrykk:

```
(define (foo cond else)
  (cond ((= cond 2) 0)
        (else (else cond))))
```

```
(define cond 3)
```

```
(define (else x) (/ x 2))
```

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

Hva blir returnert, og viktigere, *hvorfor*, når du så evaluerer følgende uttrykk:

```
(foo 2 square)
```

```
(foo 4 square)
```

```
(cond ((= cond 2) 0)
      (else (else 8)))
```

2 Primitiver / innebygde prosedyrer

- (a) Legg til prosedyrene 1+ og 1- som innebygde primitiver i evaluatoren. Prosedyrene skal ta ett argument og returnere verdien av å henholdsvis legge til eller trekke fra 1:

```
;;; MC-Eval input:
(1+ 2)
```

```
;;; MC-Eval value:
3
```

```
;;; MC-Eval input:
(1- 2)
```

```
;;; MC-Eval value:
1
```

- (b) Når vi eksperimenterer med evaluatoren kan det hende vi ønsker å legge til nye primitiver. Én måte å gjøre det på er å legge til oppslag i listen `primitive-procedures` og så evaluere uttrykket

`(set! the-global-environment (setup-environment))` på nytt, men i så fall så mister vi alle definisjoner vi allerede har lagt til i den globale omgivelsen under kjøring av evaluatoren. Definer en prosedyre `install-primitive!` som lar oss legge til nye primitive prosedyrer i den globale omgivelsen slik at de er tilgjengelige når vi re-starter REPL'en *uten* å måtte re-initialisere den globale omgivelsen på nytt med `(setup-environment)`. F.eks, etter et kall som

```
(install-primitive! 'square (lambda (x) (* x x)))
```

vil den globale omgivelsen til evaluatoren ha `square` bundet til et primitivt prosedyre-objekt (tagget med `primitive`).

3 Nye *special forms* og alternativ syntaks

- (a) Merk at evaluatoren vår er skrevet helt uten bruk av `and` og `or`. Likevel vil den kunne støtte dem i den nye versjonen av Scheme som den implementerer. Legg til `and` og `or` som nye *special forms* i evaluatoren med de nødvendige syntaks-prosedyrer og evalueringsregler som dette krever.

- (b) Implementer en ny syntaks for *if* i evaluatoren som lar oss skrive uttrykk på denne formen.

```
(if <test1>
  then <utfall1>
  elsif <test2>
  then <utfall2>
  else <utfall3>)
```

Et uttrykk kan ha vilkårlig mange *elsif*-grener (inkludert ingen) og vi regner *else* som obligatorisk.

- (c) Evaluering av såkalte *deriverte uttrykk* er basert på at man først transformerer det til et annet uttrykk i språket som så evalueres. I evaluatoren er *cond* et eksempel på et derivert uttrykk: det transformeres til en kjede av *if*-uttrykk og det er dette som egentlig blir evaluert (se bl.a. *cond*→*if*).

Foreløpig har ikke evaluatoren vår støtte for *let*, men som vi vet er *let* også et derivert uttrykk som egentlig er basert på prosedyrekall med *lambda*:

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
      :
      (<varn> <expn>))
  <body>)
```

er egentlig en forkortelse for

```
((lambda (<var1> <var2> ... <varn>)
  <body>)
  <exp1> <exp2> ... <expn>))
```

Implementer støtte for *let* basert på syntaktisk transformasjon til *lambda*-applikasjon som over. (Legg også til de nødvendige endringer i *eval-special-form*, osv.)

- (d) Hvis du har løst oppgave 3c over så har evaluatoren allerede støtte for *let*. Men nå ønsker vi å endre litt på dette slik at Scheme-varianten vår bruker en litt alternativ syntaks:

```
(let <var1> = <exp1> and
    <var2> = <exp2> and
    :
    <varn> = <expn> in
  <body>)
```

I read-eval-print-loopen skal vi altså kunne ha f.eks følgende interaksjon:

```
;;; MC-Eval input:
(let x = 2 and
  y = 3 in
  (display (cons x y))
  (+ x y))

(2 . 3)
;;; MC-Eval value:
5
```

- (e) Som vi vet kan iterative prosesser uttrykkes som vanlige prosedyrekall i Scheme. Derfor finnes det heller ingen dedikert Scheme-syntaks for å uttrykke iterasjon slik som *for*, *while*, *until*, etc. slik man har i en del andre språk. Implementer støtte for *while* som et derivert uttrykk i evaluatoren. Detaljene for nøyaktig hvordan dette skal se ut og hvordan den brukes overlates til deg, så vis også eksempler på bruk.

4 Bonusoppgave: Alternativ semantikk; dynamisk binding

- Denne oppgaven er helt *frivillig* og gir ikke uttelling i poengfordelingen. Den er bare ment som en ekstra nøtt for dem som måtte ha lyst!

Variabler i Scheme bindes *statisk*. Dette vil si at frie variabler i en prosedyre tar sin verdi fra omgivelsen der prosedyren ble definert. I noen Lisp-varianter finnes også *dynamisk* binding. Dette vil si at de frie variablene i prosedyren i stedet tar sin verdi fra omgivelsen der prosedyren *kalles*. Ta følgende sekvens av uttrykk som eksempel:

```
(define x 42)
```

```
(define (foo)  
  x)
```

```
(let ((x 24))  
  (list x (foo)))
```

I en vanlig Scheme med statisk binding så vil det siste uttrykket over evaluerer til `(24 42)`. I en Scheme med dynamisk binding derimot så vil vi få `(24 24)`.

Gjør om evaluatoren til å implementere dynamisk i stedet for statisk binding. (Tips: Endringene som skal til er forbløffende små!)

Lykke til og god koding!