Innlevering (1a) i INF2810 (Vår 2016): Løsningsforslag

1 Grunnleggende syntaks og semantikk i Scheme (2 poeng)

- Her skal vi prøve å skrive noen uttrykk til *read-eval-print loop*'en (REPL); det interaktive Scheme-promptet i programmeringsomgivelsen vår. Hvilken verdi, eller eventuelt hva slags feil, evaluerer følgende uttrykk til? Forklar kort hvorfor. Her kan det være bra å støtte seg på evalueringsreglene i seksjon 1.1.3 i SICP.

  (a) `(* (+ 4 2) 5)`

  Svar: `30`. Prosedyren `+` anvendes på `4` og `2` og evaluerer til `6`; `*` anvendes så på `6` og `5` og returnerer `30`.

  (b) `(* (+ 4 2) (5))`

  Svar: Uttrykket `(5)` tolkes som et forsøk på å kalle prosedyren `5`. Vi får en feilmelding siden `5` evaluerer til seg selv, tallet fem, og ikke en prosedyre.

  (c) `(* (4 + 2) 5)`

  Svar: I uttrykket `(4 + 2)` brukes det infiks-notasjon mens Scheme er basert på prefiks-notasjon. Siden Scheme forventer at det første liste-elementet er en prosedyre får vi en feilmelding på samme måte som i oppgaven over.

  (d) `(define bar (/ 42 2))`
  `bar`

  Svar: `bar` er bundet til `21`, verdien av $42/2$.

  (e) `(- bar 11)`

  Svar: `10`, fordi `bar` er bundet til `21`, og $21 - 11 = 10$

  (f) `(/ (* bar 3 4 1) bar)`

  Svar: `12`. Det er fortsatt slik at `bar` er bundet til `21` (subtraksjonen over modifiserte ikke `bar`). $21 \times 3 \times 4 \times 1 = 252$, og $252/21 = 12$.

2 Kontrollstrukturer og egendefinerte prosedyrer (3 poeng)

I denne oppgaven skal vi jobbe med predikater og kondisjonale uttrykk, i tillegg til å definere egne prosedyrer. Relevante seksjoner i SICP er 1.1.3, 1.1.4 og 1.1.6.

(a) Husk at logiske predikater som `or` og `and`, og kondisjonale uttrykk som `if`, ikke er vanlige prosedyrer men eksempler på såkalte *special forms*. Se på de tre uttrykkene under. Merk at de første to inneholder en syntaktisk feil og det tredje et kall på en prosedyre som ikke er definert. Hvilke verdier evalueres uttrykkene til på REPL'en? Forklar hvorfor. Se om du også klarer å forklare hvordan disse eksemplene viser at `or`, `and`, og `if` er nettopp *special forms*, og altså bryter med evalueringsregelen Scheme ellers bruker for vanlige prosedyrer (seksjon 1.1.3 i SICP er relevant her).

```
(or (= 1 2)
    "piff!"
```

```
    "paff!"
    (zero? (1 - 1)))

(and (= 1 2)
     "piff!"
     "paff!"
     (zero? (1 - 1)))

(if (positive? 42)
    "poff!"
    (i-am-undefined))
```

Svar: Det første `or`-uttrykket returnerer `"piff!"`. Forklaring: `or` er en *special form* og har dermed en litt annen semantikk enn en vanlig prosedyre der alle argumentene evalueres før vi anvender den. I stedet evalueres argumentene én og én, fra venstre til høyre, og så fort vi finner et som evaluerer til en verdi som ikke er `#f` så returneres denne og ingen flere argumenter evalueres. I uttrykket over evalueres altså aldri `(zero? (1 - 1))`, som ellers ville gitt en feilmelding.

`and`-uttrykket returnerer `#f`. Forklaring: I likhet med `or` evaluerer også `and` argumentene sine én og én, fra venstre til høyre, men så fort det finner et som evaluerer til `#f`, slik som `(= 1 2)` over, så returneres altså dette og ingen flere argumenter evalueres. I uttrykket over evalueres altså heller aldri `(zero? (1 - 1))`.

I det siste uttrykket returneres `"paff!"`. Forklaringen er stort sett lik som over: `if` er også en *special form* og følger dermed sin egen evalueringsregel. Siden test-uttrykket i `if`'en evalueres til sant returneres verdien av det neste konsekvens-uttrykket. Det siste alternativet med den udefinerte prosedyren ville bare blitt evaluert dersom testen var usann.

Evalueringsregelen som Scheme bruker for vanlige prosedyrekall går ut på å først evaluere alle uttrykkene i argumentsposisjon, og deretter anvendes prosedyren (prefikset) på verdiene av disse (dette kalles noen ganger *applicative order evaluation*). Hvis `or`, `and` eller `if` i uttrykkene over hadde oppført seg som vanlige prosdyrer så hadde vi altså fått feilmeldinger om henholdsvis syntaksfeil eller bruk av udefinerte prosedyrer. Siden de er *special forms* så har de derimot egne evalueringsregler som lar oss utsette evalueringen av uttrykk i argumentsposisjon til de eventuelt trengs.

(b) Definer en prosedyre som heter `sign` som tar et tall som argument og returnerer `-1` dersom tallet er negativt, `1` hvis det er positivt, og `0` hvis tallet er null. Skriv to versjoner; én som bruker `if` og en som bruker `cond`.

Svar:

```
(define (sign x)
  (cond ((negative? x) -1)
        ((positive? x) 1)
        (else 0)))

(define (sign x)
  (if (< x 0)
      -1
      (if (> x 0)
          1
          0)))
```

(c) Prøv å skrive en versjon av `sign` fra oppgave (b) over som hverken bruker `cond` eller `if` men istedet bruker de logiske predikatene `and` og `or`.

Svar:

```
(define (sign x)
  (or (and (< x 0) -1)
      (and (> x 0) 1)
      0))
```

3 Rekursjon, iterasjon og blokkstruktur (5 poeng)

(a) Skriv to prosedyrer `add1` og `sub1` som hver tar et tall som argument og returnerer resultatet av å henholdsvis legge til og trekke fra én. Eksempler på bruk:

```
? (add1 3) → 4
? (sub1 2) → 1
? (add1 (sub1 0)) → 0
```

(b) Basert på prosedyrene fra oppgaven over skal du nå definere en rekursiv prosedyre `plus` som tar to positive heltall som argumenter og returnerer resultatet av å legge dem sammen. Resultatet skal altså være det samme som om vi kalte den primitive prosedyren + på de to argumentene direkte, men her skal vi altså klare oss med kun `sub1` og `add1`. Husk at prosedyren din skal være *rekursiv*, dvs. at prosedyren kaller på seg selv i sin egen definisjon. (Tips: Predikatet `zero?` kan også bli nyttig. En relevant seksjon i SICP for denne og neste oppgave er 1.2.)

(c) I denne oppgaven skal vi forholde oss til skillet mellom prosedyre og prosess, og forskjellen mellom rekursive og iterative prosesser. Prøv å gi en analyse av den rekursive prosedyren du definerte i oppgave (b) over: Gir den opphav til en *iterativ* eller en *rekursiv* prosess? Forklar svaret ditt. Avhengig av hva slags type prosess den opprinnelige prosedyren din fører til, prøv å definere en ny variant som fører til den andre typen prosess.

NB: Dersom du ikke fikk til deloppgave (b) over, prøv uansett her å gi en kort forklaring på forskjellen mellom rekursive og iterative prosesser.

Svar (a–c):

```
(define (add1 x)
  (+ x 1))

(define (sub1 x)
  (- x 1))


;; gir en rekursiv prosess:

(define (plus x y)
  (if (zero? x) ;; avslutte rekursjon?
      y
      (add1 (plus (sub1 x) y))))
```

```
;; gir en iterativ prosess:

(define (plus x y)
  (if (zero? x) ;; avslutte rekursjon?
      y
      (plus (sub1 x) (add1 y))))
```

(d) Ta en titt på prosedyren `power-close-to` som er definert under.

```
(define (power-close-to b n)
  (power-iter b n 1))

(define (power-iter b n e)
  (if (> (expt b e) n)
       e
      (power-iter b n (+ 1 e))))
```

Prosedyren tar to argumenter `b` og `n`, og returnerer det minste heltallet x slik at $\texttt{b}^x > \texttt{n}$. For eksempel vil et kall som `(power-close-to 2 8)` returnere `4` (fordi $2^4 > 8$). Videre ser vi også at den benytter seg av hjelpe-prosedyren `power-iter`. Skriv om disse prosedyrene slik at de bruker *blokkstruktur*, altså at `power-iter` er definert internt i definisjonen til `power-close-to`. Se om du samtidig da klarer å forenkle definisjonen av hjelpe-prosedyren, og prøv å kort forklare hvordan/hvorfor det lar seg gjøre. Relevant seksjon i SICP er 1.1.8.

Svar:

```
(define (power-close-to b n)
  (define (power-iter e)
    (if (> (expt b e) n)
        e
        (power-iter (+ 1 e))))
  (power-iter 1))
```

(e) Seksjon 1.2.2 i SICP inneholder en definisjon av en iterativ prosedyre for å beregne Fibonacci-tall. Definisjonen er gjengitt under.

```
(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
    b
    (fib-iter (+ a b) a (- count 1))))
```

Vi ser at prosedyren `fib` bruker den iterative hjelpefunksjonen `fib-iter`. Er det mulig å forenkle den interne definisjonen av `fib-iter` nar man skriver om til å bruke blokkstruktur? Begrunn svaret.

Svar: I motsetning til deloppgave (d) over er det ikke mulig å forenkle parameterlisten til `fib-iter`, dvs. gjøre om bundne til frie variabler i hjelpeprosedyren. Dette fordi alle parametrene (`a`, `b`, og `count`) får nye verdier i hvert rekursive kall.