Innlevering 1a i IN2040 (Høst 2021)

- Det er mulig å få 10 poeng tilsammen for denne innleveringen 1a, som altså er del én av to for obligatorisk oppgave 1. Du må ha minst 12 poeng tilsammen på 1a pluss 1b for å få godkjent. Tematikken denne gang er stoff som dekkes i de to første forelesningene. Oppgavene skal løses individuelt og leveres via Devilry (som én enkelt fil) *innen fredag, 10. september, kl. 23:59*. Husk at filer med Scheme-kode skal ha ekstensjonen `.scm`. Du finner mer informasjon om innleverings-systemet i IN2040 på emnets semesterside: `https://www.uio.no/studier/emner/matnat/ifi/IN2040/h21/innleveringer.html`
- For å inkludere *kommentarer* i kildekoden bruker man semikolon. Alt som står etter '`;;`' på samme linjen ignoreres av Scheme, og det er alltid fint om du dokumenterer koden din med rikelig med kommentarer.
- Husk at dersom du har spørsmål underveis kan du:
  - få hjelp av gruppelærerene på gruppetimen;
  - poste spørsmål på GitHub-diskusjonsforumet: `https://github.uio.no/IN2040/h21/issues`;
  - eller sende epost til `in2040-hjelp@ifi.uio.no`.

  Som supplement til SICP (læreboka) er det også mye nyttig informasjon å finne i Scheme-dokumentasjonen: `http://groups.csail.mit.edu/mac/ftpdir/scheme-reports/r5rs-html/r5rs_toc.html`.

1 Grunnleggende syntaks og semantikk i Scheme

- 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)`

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

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

  (d)
  ```
  (define bar (/ 44 2))
  bar
  ```

  (e) `(- bar 11)`

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

2 Kontrollstrukturer og egendefinerte prosedyrer

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 uttrykk 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)
    "paff!"
    "piff!"
    (zero? (1 - 1)))

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

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

(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`.

(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`.

3 Rekursjon, iterasjon, og blokkstruktur

(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.

(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 hvorfor og hvordan det lar seg gjøre. Relevant seksjon i SICP er 1.1.8.

(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` når man skriver om til å bruke blokkstruktur? Begrunn svaret.

Lykke til og god koding!