Innlevering 1b i INF2810, vår 2017

- Det er mulig å få opptil 10 poeng for denne innleveringen (1b), som altså er andre og siste del av obligatorisk oppgave 1. Du må ha minst 12 poeng tilsammen for 1a pluss 1b for å få godkjent den. For mer informasjon, se kurssiden: http://www.uio.no/studier/emner/matnat/ifi/INF2810/v17/innleveringer.html.
- Oppgavene skal løses individuelt og leveres via Devilry innen *fredag, 24. februar, kl 12:00 (om morgenen)*.
- Trenger du hjelp? Spør i gruppetimene eller forumet: https://piazza.com/uio.no/spring2017/inf2810/.

1 Par og lister

- For å få på plass en solid forståelse av de grunnleggende datatypene *par* og *lister* i Scheme skal vi her tegne 'boks-og-peker'-diagrammer for å vise strukturen til noen enkle eksempler. Tenk at uttrykkene under skrives inn på REPL'et, og tegn så strukturen for returverdien. Bruk gjerne ASCII-grafikk som kan inkluderes som kommentar i samme fil som Scheme-kode. En annen mulighet er å bruke favorittegneprogrammet ditt eller ta bilde av en papirtegning med mobilen og levere dette som en egen fil. I denne oppgaven kan det være bra å støtte seg på seksjon 2.2 og 2.2.1 i SICP.

  (a) `(cons 47 11)`

  (b) `(cons 47 '())`

  (c) `(list 47 11)`

  (d) `'(47 (11 12))`

  (e)
  ```
  (define foo '(1 2 3))
  (cons foo foo)
  ```

- Vis hvordan man trekker ut elementet `42` fra følgende lister, bare ved bruk av `car` og `cdr`:

  (f) `(0 42 #t bar)`

  (g) `((0 42) (#t bar))`

  (h) `((0) (42 #t) (bar))`

  (i) Vis hvordan listen fra deloppgaven (g) over kan lages bare ved bruk av prosedyren `cons` *og* bare ved bruk av `list`.

2 Rekursjon over lister og høyereordens prosedyrer

(a) På forelesningen den 2. februar så vi på en rekursiv egendefinert variant av den innebygde prosedyren `length`. Prosedyren tar en liste som argument og teller hvor mange elementer den inneholder. Skriv en *halerekursiv* versjon av denne (kall den f.eks. `length2`).

(b) På forelesningen torsdag 2. februar snakket vi om en høyerordens prosedyre som vi kalte `reduce` og som vi definerte som følger:

```
(define (reduce proc init items)
  (if (null? items)
      init
      (proc (car items)
            (reduce proc init (cdr items)))))
```

Kalleksempler:

```
? (reduce + 0 '(1 2 3)) → 6

? (reduce cons '() '(1 2 3)) → (1 2 3)
```

Resultatet av det siste kalleksemplet blir altså at vi får returnert en kopi av den opprinnelige lista, og vi kan tenke at rekursjonen 'ekspanderes' som følger:

```
(cons 1 (cons 2 (cons 3 '())))
```

Her skal vi derimot definere en variant vi kan kalle `reduce-reverse`, som skal gi følgende resultat:

```
? (reduce-reverse cons '() '(1 2 3)) → (3 2 1)
```

Vi kan tenke at rekursjonen skal ekspanderes som

```
(cons 3 (cons 2 (cons 1 '())))
```

Definer prosedyren `reduce-reverse`. Oppgi hvorvidt prosedyredefinisjonen din er halerekursiv eller ikke, og i tillegg hva slags prosess den gir opphav til.

(c) Skriv et høyereordens predikat `all?` som tar et annet predikat og en liste som argument og returnerer `#t` dersom alle elementer i lista tester sant for predikatet og `#f` ellers. Dersom vi gir den tomme lista som argument kan den returnere `#t`. Kalleksempler:

```
? (all? odd? '(1 3 5 7 9)) → #t

? (all? odd? '(1 2 3 4 5)) → #f

? (all? odd? '()) → #t
```

Vis også hvordan `all?` kan kalles med en anonym prosedyre (`lambda`-uttrykk) definert slik at `all?` returnerer `#f` dersom noen av tallene i listeargumentet er høyere enn 10. Kalleksempler (der du skal fylle inn uttrykket for `????`):

```
? (all? ???? '(1 2 3 4 5)) → #t

? (all? ???? '(1 2 3 4 50)) → #f
```

(d) Skriv en prosedyre `nth` som tar to argumenter: et (indeks)tall og en liste. `nth` skal returnere liste-elementet på posisjonen som indekstallet indikerer. Vi ønsker å telle fra `0`, dvs. at det første elementet har indeks `0`. (Du trenger ikke bry deg om feilsjekking, vi bare antar at prosedyren ikke vil kalles med en ugyldig indeks.) Eksempel på kall:

```
? (nth 2 '(47 11 12 13)) → 12
```

(e) Skriv en prosedyre `where` som tar to argumenter: et tall og en liste. `where` beregner posisjonsindeks til (første forekomst av) tallet i listen, f.eks.

```
? (where 3 '(1 2 3 3 4 5 3)) → 2

? (where 0 '(1 2 3 3 4 5 3)) → #f
```

(f) I forelesningen ga vi en definisjon av prosedyren `map` som var noe forenklet (sammenliknet med den innebygde R5RS-versjonen av prosedyren) ved at vår versjon bare kan operere over én liste om gangen. Skriv en variant `map2` som opererer over to lister parallelt, dvs. kaller dens første argument (som må være en prosedyre) på de første elementene i listene, så på elementene på andre plass, osv. Hvis de to listene ikke har samme antall elementer så avslutter `map2` når den har kommet gjennom den korteste listen, f.eks.

```
? (map2 + '(1 2 3 4) '(3 4 5)) → (4 6 8)
```

(g) I eksempelet over brukte vi den forhåndsdefinerte Scheme-prosedyren `+` som første argument til `map2`. Vi skal nå i stedet ta i bruk anonyme `lambda`-prosedyrer for å operere over listeelementer. Vis hvordan `map2` kan kombineres med en anonym prosedyre som beregner gjennomsnittet av to tall. Med argumentene `'(1 2 3 4)` og `'(3 4 5)` burde returverdien bli `(2 3 4)`.

(h) Skriv en prosedyre `both?` som genererer et predikat som tar to argumenter. `both?` selv tar ett argument som er et predikat som kun tar ett argument. Returverdien til `both?` er en prosedyre som returnerer `#t` i tilfelle predikatet som var argument til `both?` er gyldig for begge argumentene. Her er noen eksempler som viser hva vi er ute etter:

```
? (map2 (both? even?) '(1 2 3) '(3 4 5)) → (#f #t #f)

? ((both? even?) 2 4) → #t

? ((both? even?) 2 5) → #f
```

(i) Skriv en prosedyre `self` som tar en prosedyre som argument og returnerer en ny prosedyre slik at f.eks.

```
? ((self +) 5) → 10

? ((self *) 3) → 9

? (self +) → #<procedure>

? ((self list) "hello") → ("hello" "hello")
```

Lykke til og god koding!