Innlevering 2a
INF2810, vår 2015

- Det er mulig å få 10 poeng tilsammen for denne innleveringen (2a), og man må ha minst 12 poeng tilsammen (av 20 mulige) for 2a + 2b.

- Vi anbefaler at oppgavene fra nå av løses i grupper på to eller tre studenter sammen. Hvis du heller ønsker å levere individuelt er det likevel ok. For gruppeleveringer: Registrer gruppen i Devilry ved levering, men (NB) la også første linje i kodefilen være en kommentar som lister opp hvem som er med på gruppa. Hvis du ikke har laget prosjektgruppe i Devilry før kan du se på dokumentasjonen her:

  `https://devilry-userdoc.readthedocs.org/en/latest/student_groups.html`

- Koden leveres som én enkelt *.scm*-fil via Devilry *innen fredag 20.03 kl 14:00*. Ta kun med din egen kode, ikke prekoden (med mindre du har gjort endringer). Tekstsvar kan gjerne inkluderes som kommentarer i kodefilen.

- Hovedtematikken denne gang er huffmankoding, som ble dekket i 6. forelesning (20. feb) og i seksjon 2.3.4 i SICP. Det er viktig å ha lest denne seksjonen før dere begynner på oppgaven. Oppgavene bygger videre på abstraksjonsbarrieren for huffmantrær som er definert i SICP, der trær er implementert som lister. Koden for dette finner dere i fila '*huffman.scm*' på semestersiden:

  `www.uio.no/studier/emner/matnat/ifi/INF2810/v15/undervisningsmateriale/huffman.scm`

- Merk at prekoden gjør bruk av diverse innebygde kortformer for kjeder av `car` og `cdr`. Et uttrykk som `(cadr x)` er det samme som `(car (cdr x))`, og `(caadr x)` tilsvarer `(car (car (cdr x)))` osv. F.eks:

```
? (define foo '(1 (2 3) 4 5 6))
? (cadr foo) → (2 3)
? (caddr foo) → 4
? (cdadr foo) → (3)
```

1 Diverse

Før vi begynner med huffmankoding tar vi med et par mindre oppgaver der vi skal bryne oss litt på høyereordens prosedyrer, og `lambda` og `let`.

(a) I forelesningen 12. februar så vi på en implementasjon av datatypen par ("cons-celler") som en prosedyre. Her er nok en alternativ variant av par representert som prosedyre, gitt ved følgende konstruktor;

```
(define (p-cons x y)
  (lambda (proc) (proc x y)))
```

Definer selektorene `p-car` og `p-cdr` for denne representasjonen. Vi bruker `p-*` i navnene til grensesnittet vårt så de ikke kolliderer med de innebygde versjonene. Ellers vil vi at `p-car` og `p-cdr` skal gi samme resultat som `car` og `cdr` gjør for den innebygde versjonen av `cons`. Noen kalleksempler:

```
? (p-cons "foo" "bar")
→ #<procedure>

? (p-car (p-cons "foo" "bar"))
→ "foo"

? (p-cdr (p-cons "foo" "bar"))
→ "bar"

? (p-car (p-cdr (p-cons "zoo" (p-cons "foo" "bar"))))
→ "foo"
```

(b) Vis hvordan de to `let`-uttrykkene under kan skrives om til `lambda`-uttrykk. (Forelesningsnotatene fra 12/2 er relevante her.) Oppgi også hvilken verdi uttrykkene evaluerer til, og forklar helt kort hvorfor det andre `let`-uttrykket evaluerer til en annen verdi enn det første.

```
? (define foo 30)


? (let ((x foo)
        (y 20))
    (+ foo x y))



? (let ((foo 10))
    (let ((x foo)
          (y 20))
      (+ foo x y)))
```

(c) Anta at vi skriver inn følgende sekvens på REPL-promptet:

```
? (define a1 (list 1 2 3 4))
? (define a2 (list + - * /))
? (define a3 (list 5 6 7 8))
? (map (lambda (x y z) (y x z))
       a1 a2 a3)
```

Hva blir resultatet? Forklar med en setning eller to hva som skjer i kallet på `map` over. Vis også et eksempel på hvordan den anonyme prosedyren `(lambda (x y z) (y x z))` kan kalles direkte (uten at den inngår i `map`), du må altså selv velge eksempler på passende argumenter og vise hvordan et kall kan se ut og hva resultatet vil bli.

2 Huffmankoding

(a) Først skal vi skrive en enkel hjelpeprosedyre som vi kan få bruk for senere. Det finnes innebygde prosedyrer med liknende funksjonalitet, men vi skal skrive den selv for øvelsens skyld.

Skriv et predikat `member?` som tar et symbol og en liste som argument og returnerer sant dersom symbolet finnes i lista og usant ellers. Det innebygde predikatet `eq?` kan brukes for teste for identitet for symboler. Eksempel på kall:

```
? (member? 'forest '(lake river ocean)) → #f
? (member? 'river '(lake river ocean)) → #t
```

(b) Gjør deg kjent med koden i definisjonen til `decode` i '*huffman.scm*'. Skriv et par setninger om hva som er vitsen med den interne prosedyren `decode-1` – den kalles jo med de samme argumentene som hovedprosedyren så hvorfor ikke heller bare kalle denne?

(c) Skriv en halerekursiv versjon av `decode`.

(d) Fila '*huffman.scm*' inneholder et eksempel på et kodetre og en bitkode, henholdsvis bundet til variablene `sample-tree` og `sample-code`. Hva er resultatet av å kalle prosedyren `decode` fra oppgaven over med disse som argument?

```
? (decode sample-code sample-tree) → ?
```

(e) Skriv en prosedyre `encode` som transformerer en sekvens av symboler til en sekvens av bits. Input er en liste av symboler (meldingen) og et huffmantre (kodeboken). Output er en liste av `0` og `1`. Du kan teste prosedyrene dine med å kalle `decode` på returverdien fra `encode` og sjekke at du får samme melding. F.eks:

```
? (decode (encode '(ninjas fight ninjas) sample-tree) sample-tree)
→ (ninjas fight ninjas)
```

- Litt bakgrunn for oppgave *(f)* under: Seksjonene *Generating Huffman trees* og *Sets of weighted elements* under 2.3.4 i SICP beskriver en algoritme for å generere Huffman-trær. Det samme dekkes på foilene fra forelesningen 20/2. Algoritmen opererer på en mengde av noder der vi suksessivt slår sammen de to nodene som har lavest frekvens. Mengden av noder er implementert som en liste. Som diskutert på forelesningen så kan algoritmen utføres mest effektiv dersom vi sørger for å holde nodelista sortert etter frekvens. SICP-koden i '*huffman.scm*' inneholder funksjonalitet for å generere en sortert liste av løvnoder som kan brukes for å initialisere algoritmen; `adjoin-set` og `make-leaf-set`. For eksempel:

```
? (make-leaf-set '((a 2) (b 5) (c 1) (d 3) (e 1) (f 3)))
→ ((leaf e 1) (leaf c 1) (leaf a 2) (leaf f 3) (leaf d 3) (leaf b 5))
```

(f) Skriv en prosedyre `grow-huffman-tree` som tar en liste av symbol/frekvens-par og returnerer et huffmantre. Eksempel på kall:

```
? (define freqs '((a 2) (b 5) (c 1) (d 3) (e 1) (f 3)))
? (define codebook (grow-huffman-tree freqs))
? (decode (encode '(a b c) codebook) codebook) → (a b c)
```

(g) Vi er gitt følgende alfabet av symboler med frekvenser oppgitt i parentes:

ninjas (57), *samurais* (20), *fight* (45), *night* (12), *hide* (3), *in* (2), *ambush* (2), *defeat* (1), *the* (5), *sword* (4), *by* (12), *assassin* (1), *river* (2), *forest* (1), *wait* (1), *poison* (1).

Generer et huffmantre for alfabetet på bakgrunn av frekvensene, og svar så på de følgende spørsmålene. Hvor mange bits bruker det på å kode følgende melding? Hva er den gjennomsnittlige lengden på hvert kodeord som brukes? (Vi tenker at alle symbolene representeres i én og samme liste slik at linjeskift ignoreres.) Til slutt: hva er det minste antall bits man ville trengt for å kode meldingen med en kode med fast lengde (*fixed-length code*) over det samme alfabetet? Begrunn kort svaret ditt.

ninjas fight
ninjas fight ninjas
ninjas fight samurais
samurais fight
samurais fight ninjas
ninjas fight by night

(h) Skriv en prosedyre `huffman-leaves` som tar et huffmantre som input og returnerer en liste med par av symboler og frekvenser, altså det samme som vi kan bruke som utgangspunkt for å generere treet. For eksempel, for `sample-tree` i '*huffman.scm*':

```
? (huffman-leaves sample-tree)
→ ((ninjas 8) (fight 5) (night 1) (by 1))
```

(i) Den forventede gjennomsnittlige lengden på kodeordene som genereres av et huffmantre kan uttrykkes som

$$\sum_{i=1}^{n} p(s_i) \times |c_i|$$

der $p(s_i)$ er sannsynligheten for det i-ende symbolet i et alfabet med n symboler og $|c_i|$ står for lengden på kodeordet til s_i. (Sagt med andre ord: $|c_i|$ er antall bits som huffmantreet bruker på å kode symbolet s_i.) Sannsynligheten for et symbol, $p(s_i)$, er gitt ved dets relative frekvens, altså frekvensen for symbolet delt på total frekvens for alle symboler. Skriv en prosedyre `expected-codeword-length` som tar et huffmantre som input og beregner formelen over. Kalleksempel:

```
? (expected-code-length sample-tree) → 1 3/5
```

I oppgave (g) over spurte vi bl.a. om gjennomsnittslengden på kodeordene som ble brukt for å kode en melding. Mer generelt: hvilke kriterier må foreligge for at den faktiske gjennomsnittlige lengden på kodeordene i en melding tilsvarer den forventede gjennomsnittslengden slik den er definert over?