Innlevering 2a i INF2810, vår 2017

- Hovedtematikken denne gang er Huffman-koding, som ble dekket i 6. forelesning (23. februar) 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 Huffman-trær som er definert i SICP, der trær er implementert som lister. Koden for dette finner dere i fila *huffman.scm* på semestersiden:

  `http://www.uio.no/studier/emner/matnat/ifi/INF2810/v17/oppgaver/huffman.scm`

- Ta kun med din egen kode ved levering, ikke prekoden (med mindre du har gjort endringer). Du kan laste inn prekoden med prosedyrekallet `(load "huffman.scm")` øverst i din egen `.scm`-fil. Da må `huffman.scm` ligge i samme mappe som koden din. Dette gir samme effekt som om hele prekoden sto øverst i din egen fil.

- 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 pluss 2b.

- Besvarelsen leveres som én enkelt `.scm`-fil via Devilry innen *fredag, 17. mars, kl. 15:00*. Tekstsvar kan tas med som kommentarer (linjer som begynner mer `;;`), husk å oppgi oppgavenumre. Hvis dere veldig gjerne vil kan dere heller levere tekstsvar i en egen tekstfil eller PDF ved siden av.

- Vi anbefaler at oppgavene fra nå av (og ut semesteret) løses i grupper på to eller tre studenter sammen (se nærmere instruksjoner under). Selv om vi oppfordrer til gruppearbeid er det likevel mulig å levere individuelt dersom du heller ønsker det.

- Ang. levering for dem som samarbeider i gruppe:
  - La første linje i kodefilen være en kommentar som lister opp hvem som er med på gruppa.
  - Gruppa må også registreres i Devilry. En på gruppa må invitere de andre gruppemedlemmene før besvarelsen leveres i Devilry, men kun én på gruppa trenger å levere. Les mer om hvordan dere gjør dette i dokumentasjonen her::

    `https://devilry-userdoc.readthedocs.org/en/latest/student_groups.html`
  - Merk at det ikke er anledning til å endre grupper mellom 2a og 2b, eller mellom 3a og 3b. I mellom 2 og 3 er det imidlertid mulig å gjøre endringer.

- Til slutt tar vi med en påminnelse om IFI-reglementet for innleveringer, som vi også lenker til fra emnesiden: `http://www.uio.no/studier/eksamen/obligatoriske-aktiviteter/mn-ifi-oblig.html`

1 Diverse

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

(a) I forelesningen den 9. 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 så de ikke kolliderer med de innebygde versjonene (men dere skal ikke bruke dem her). 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 applikasjon av `lambda`-uttrykk. (Forelesningsnotatene fra 9/2 er relevante her.) Oppgi også hvilken verdi uttrykkene evaluerer til.

```
? (define foo 42)


? (let ((foo 5)
        (x foo))
    (if (= x foo)
        'same
        'different))


? (let ((bar foo)
        (baz 'towel))
    (let ((bar (list bar baz))
          (foo baz))
      (list foo bar)))
```

(c) Her skal dere skrive en prosedyre `infix-eval`, som skal ta ett argument `exp` som forventes å være en liste av tre elementer: en operand, en operator og nok en operand. Returverdien til `infix-eval` skal være resultatet av å anvende operatoren på operandene. Kalleksempler:

```
? (define foo (list 21 + 21))

? (define baz (list 21 list 21))

? (define bar (list 84 / 2))

? (infix-eval foo) → 42

? (infix-eval baz) → (21 21)

? (infix-eval bar) → 42
```

(d) Gitt prosedyren din `infix-eval` fra oppgaven over, hva blir resultatet av følgende kall? Eksemplet likner tilsynelatende på det siste kallet over med `bar` som argument. Forklar kort hvorfor utfallet blir annerledes?

```
? (define bah '(84 / 2))

? (infix-eval bah) → ??
```

2 Huffman-koding

(a) Først skal vi skrive en enkel hjelpeprosedyre som vi kan få bruk for senere. Skriv et predikat `member?` som tar tre argumenter: et likhetspredikat, et element og en liste: Den skal returnere `#t` dersom elementet finnes i lista og `#f` ellers. Det finnes innebygde prosedyrer med liknende funksjonalitet, men her skal vi skrive en mer generell variant som lar oss spesifisere hvilket predikat som skal brukes for å teste for likhet, f.eks. `=`, `string=?`, `eq?`, `equal?`, osv.[1] Kalleksempler:

```
? (member? eq? 'zoo '(bar foo zap)) → #f

? (member? eq? 'foo '(bar foo zap)) → #t

? (member? = 'foo '(bar foo zap)) → error: expected number

? (member? = 1 '(3 2 1 0))  → #t

? (member? eq? '(1 bar)
           '((0 foo) (1 bar) (2 baz))) → #f

? (member? equal? '(1 bar)
           '((0 foo) (1 bar) (2 baz)))  → #t
```

(b) Gjør deg kjent med koden i definisjonen til `decode` i vår fil `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 Huffman-tre (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:

[1] På slutten av forelesningen 23/2 snakket vi om forskjellen på de to sistnevnte. Grovt sagt kan vi si at `equal?` gir sant for ting som ser likt ut når det printes på REPL'et og kan bla. brukes for å sammenlikne lister ved at det rekursivt tester for strukturell ekvivalens. Predikatet `eq?` er mer strikt og sjekker om argumentene peker til identiske adresser i minnet. Merk at `eq?` kan brukes for sammenlikne symboler siden disse er konstanter i Scheme.

```
? (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 den 1. mars. 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 Huffman-tre. 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:

samurais (57), *ninjas* (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 Huffman-tre for alfabetet på bakgrunn av frekvensene, og svar så på de følgende spørsmålene med utgangspunkt i meldingen under. (Merk at denne skal forstås som *en* melding bestående av 17 symboler; linjeskiftene er ikke en del av meldingen.)

- Hvor mange bits bruker det på å kode meldingen?
- 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 Huffman-tre 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 Huffman-tre 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 Huffman-treet 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 Huffman-tre som input og beregner formelen over. Kalleksempel:

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

Lykke til, og god koding!