Innlevering (3a) i INF2810, vår 2017

- Dette er innlevering (3a); del én av den tredje obligatoriske oppgaven i INF2810. Man må ha minst 12 poeng tilsammen for (3a) + (3b), og man kan få opptil 10 poeng på hver innlevering.
- Som for oblig 2 kan oppgavene løses gruppevis (med mindre du har avtale om individuell levering). Det holder at én på hver gruppe leverer, men si i fra klart og tydelig i begynnelsen av besvarelsen hvem andre som er med på gruppa (navn + brukernavn). Svarene må leveres via Devilry *innen kl. 15:00 på fredag 28. april*.
- Husk som vanlig også å kommentere koden (med `;;`) så det blir lettere for de som skal rette å skjønne hvordan dere har tenkt. Ta også med relevante kjøringseksempler (kommentert ut) som viser at koden fungerer.
- Arkivet '`hjelpekode3a.zip`' inneholder en del hjelpeprosedyrer (i fila '`prekode3a.scm`') som dere kan bruke om dere vil:

  http://www.uio.no/studier/emner/matnat/ifi/INF2810/v17/oppgaver/hjelpekode3a.zip

  Koden inkluderer datatyper for strømmer og et utvalg listeoperasjoner tilpasset strømmer. Hvis du heller vil lage dine egne implementasjoner av noe fra '`prekode3a.scm`' må du gjerne gjøre det, men husk da å legge ved koden så retterne fortsatt kan kjøre løsningene dine (og ellers leverer du kun din egen kode). Du kan laste inn prekoden med `(load "prekode3a.scm")` øverst i kildefilen din (gitt at filene ligger i samme mappe).

  Videre inneholder arkivet to tekstfiler med engelsk tekst ('`brown.txt`' og '`wsj.txt`') som er inndata til deloppgave (2) under.
- Det er gjerne en god ide å lese gjennom hele oppgaveteksten før man setter i gang med koding.

1 Strømmer

I denne oppgaven får du bruk for strømimplementasjonen fra '`prekode3a.scm`'. Merk at hjelpeprosedyren `show-stream` skriver ut de n første elementene av en strøm og kan være nyttig for debugging.

(a) Her skal det skrives prosedyrer for å konvertere strømmer til lister og motsatt. `list-to-stream` skal ta en liste som argument og returnere en strøm av elementene i lista. `stream-to-list` skal ta en strøm som argument og returnere en liste med elementene i strømmene. I tillegg skal `stream-to-list` også ta et valgfritt argument som angir hvor mange elementer fra strømmen som skal tas med i listen, slik at prosedyren også kan brukes med uendelige strømmer. Se kalleksempler under (`stream-interval` og den uendelige strømmen av naturlige tall `nats` er definert i prekoden):

```
? (list-to-stream '(1 2 3 4 5))
→ (1 . #<promise>)
? (stream-to-list (stream-interval 10 20))
→ (10 11 12 13 14 15 16 17 18 19 20)
? (show-stream nats 15)
⇝ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...
? (stream-to-list nats 10)
→ (1 2 3 4 5 6 7 8 9 10)
```

(b) SICP gir en versjon av `stream-map` definert for én input-strøm (seksjon 3.5.1). Fullfør definisjonen under

for å lage en generalisert versjon av `stream-map` som tar én eller flere strømmer som argument (på samme måte som den innebygde `map` for lister). Merk at en liten utfordring her gjelder hvordan vi definerer *basistilfellet*, siden vi kan tenke oss at input-strømmene kan være av forskjellig lengde. For å få full uttelling på poeng her skal prosedyren defineres til å avslutte rekursjonen så fort én av input-strømmene er tomme. (Du kan gjerne ta i bruk egendefinerte hjelpeprosedyrer for dette om du synes det hjelper.) Hvis du synes dette blir for vanskelig kan du heller gjøre en forenkling ved å anta at input-strømmene er like lange og så definere basistilfellet ut fra dette.

```
(define (stream-map proc . argstreams)
  (if <???>
      the-empty-stream
      (<???>
        (apply proc (map <???> argstreams))
        (apply stream-map
               (cons proc (map <???> argstreams))))))
```

(c) Den følgende prosedyren returnerer en liste med alle distinkte symboler i en gitt liste med symboler. (Med andre ord, hvert symbol forekommer kun én gang i returlista, uansett om det forekommer én eller flere ganger i input-lista.)

```
(define (remove-duplicates lst)
  (cond ((null? lst) null)
        ((not (memq (car lst) (cdr lst)))
         (cons (car lst) (remove-duplicates (cdr lst))))
        (else (remove-duplicates (cdr lst)))))
```

`remove-duplicates` sjekker om `car` av lista er et element i `cdr` av lista. I så fall ignorerer den car og returnerer resultatet av å fjerne duplikater fra `cdr`. Ellers `cons`'er den car på dette resultatet. `remove-duplicates` bruker prosedyren `memq` fra SICP 2.3.1 (innebygd) som tester om et gitt symbol forekommer i en gitt liste. Denne kan defineres slik:

```
(define (memq item x)
  (cond ((null? x) #f)
        ((eq? item (car x)) x)
        (else (memq item (cdr x)))))
```

Petter Smart foreslår at for å tilpasse `remove-duplicates` for å fungere med strømmer så trenger vi bare modifisere `remove-duplicates` og `memq` ved å bytte ut `cons` med `cons-stream`, `car` med `stream-car`, `cdr` med `stream-cdr`, og `null?` med `stream-null?`. Kan du se et potensielt problem med Petter Smarts forslag? Forklar kort.

(d) Vi skal her se nærmere på implementasjonen av utsatt evaluering i Scheme, som diskutert i forelesningen den 30. mars. Vi skal ta utgangspunkt i følgende prosedyre som ikke gjør annet enn å returnere argumentet sitt etter først å ha printet det.

```
(define (show x)
  (display x)
  (newline)
   x)
```

Hva vises på REPL'et når følgende uttrykk evalueres (i gitt rekkefølge)? Forklar kort hvorfor.

```
? (define x
    (stream-map show
                (stream-interval 0 10)))
```

```
? (stream-ref x 5)

? (stream-ref x 7)
```

(Merk at både `show` og `stream-interval` er inkludert i prekoden. Det samme er `stream-ref` som fungerer liknende `list-ref` for lister og returnerer det n-te elementet i sekvensen.)

2 Språkmodeller (basal 'maskinlæring')

En språkmodell (*language model*) er en enkel statistikk over ordfrekvenser, dvs. sannsynligheter for at et bestemt ord brukes i en bestemt kontekst. Holder man seg til ikke-dadaistisk engelsk for eksempel, er det betydelig mer sannsynlig at ordet *jury* etterfølger ordet *the*, enn at ordet *this* skulle etterfølge *the*. Språkmodeller har mange anvendelser, for eksempel i prediktivt tekstinnputt på mobil, i automatisk oversettelse fra et språk til et annet, eller i talegjenkjenning: *It's hard to recognize speech* og *It's hard to wreck a nice beach*, for eksempel, er veldig nære hverandre i uttalen, men den første ordsekvensen er nok betydelig mer sannsynlig. I denne oppgaven skal vi implementere enkle språkmodeller på en måte som skalerer til større tekstmengder.

Gitt en setning (dvs. sekvens av ord) $s = w_1, w_2, \ldots, w_n$ vil vi gjøre antakelsen om at sannsynligheten til ordet w_i bare avhenger av ordet som star rett foran, dvs. w_{i-1}. Dette kan formaliseres som $P(w_i|w_{i-1})$, eller den betingete *sannsynligheten* ('P') for w_i *gitt* ('|') w_{i-1}. Med utgangspunkt i sannsynlighetsbegrepet for enkeltord i kontekst, kan sannsynligheten for hele setningen s beregnes som

$$\begin{aligned} P(s) &= P(w_1|\texttt{<s>}) \times P(w_2|w_1) \times \ldots \times P(w_n|w_{n-1}) \times P(\texttt{</s>}|w_n) \\ &= P(w_1|\texttt{<s>}) \times P(\texttt{</s>}|w_n) \times \textstyle\prod_{1 \le i \le n} P(w_i|w_{i-1}) \end{aligned} \tag{1}$$

Her brukes symbolene `<s>` og `</s>` som 'pseudoord' for å markere begynnelsen og slutten til setningen, dvs. markører for de hypotetiske posisjonene w_0 og w_{n+1} rett foran og rett bak setningen.

For å estimere (dvs. 'lære') enkeltsannsynlighetene $P(w_i|w_{i-1})$ må vi bare telle opp i løpende tekst hvor ofte w_{i-1} etterfølges av w_i; denne opptellingen gir altså frekvensen til ordparet $\langle w_{i-1}, w_i \rangle$, som så kan settes i forhold til hvor ofte enkeltordet w_{i-1} brukes totalt.

For eksempel kan vi tenke oss at en stor tekstsamling viser at ordparene $\langle$*the*, *jury*$\rangle$ og $\langle$*the*, *this*$\rangle$ brukes henholdsvis 250 og 0 ganger; videre finnes ordet *the* totalt 1000 ganger i samme tekstsamlingen (uansett hva som måtte følge etter). Da blir $P(\textit{jury}|\textit{the}) = \frac{250}{1000} = 0.25$, mens $P(\textit{this}|\textit{the}) = \frac{0}{1000} = 0$. Selv om man trener på store tekstmengder vil det ofte være tilfelle at en setning inneholder ordpar som er ukjent for språkmodellen; strengt tatt ville da også produktet av bigramsannsynligheten for hele setningen, $P(s)$, bli null. For å unngå dette kan man bruke en veldig liten sannsynlighet som standardverdi ('fall-back') for ukjente ordpar, for eksempel $\frac{1}{N}$, der N er det totale antall ord i tekstsamlingen.

(a) I første delen av denne oppgaven skal vi designe og implementere en datastruktur som kan brukes for å telle opp frekvenser til ordpar (som vi kaller for *bigrammer*)—altså en type todimensjonal assosiativ tabell. Grensesnittet til tabellen skal inneholde prosedyrene:

- `make-lm` — returnerer en tom språkmodell;
- `lm-lookup-bigram` *lm string*$_1$ *string*$_2$ — returnerer verdien (rå frekvens eller betinget sannsynlighet) for ordparet $\langle$*string*$_1$, *string*$_2\rangle$ i språkmodellen *lm*;
- `lm-record-bigram!` *lm string*$_1$ *string*$_2$ — inkrementerer telleren til ordparet $\langle$*string*$_1$, *string*$_2\rangle$ i språkmodellen *lm*, samt telleren til enkeltordet *string*$_1$; hvis enkeltordet eller paret ikke finnes i *lm* fra før legger `lm-record-bigram!` disse til med initial frekvens 1.

(b) Når vi først har språkmodelldatastrukturen klar kan vi lese inn tekst og foreta frekvensopptellingen. Vårt arkiv '`hjelpekode3a.zip`' inneholder en prosedyre `read-corpus`, samt to tekstfiler '`brown.txt`' og '`wsj.txt`'; de kan brukes som

```
(define brown (read-corpus "brown.txt"))
```

`read-corpus` returnerer en liste med setninger, der hver setning er en liste av ord (inkludert start- og sluttmarkørene); vi representerer ord som Scheme-strenger, for eksempel `"this"` eller `"<s>"`. Strenger kan blant annet sammenliknes med `string=?` eller `string<=?`.

For å estimere bigramsannsynligheter, skriv en prosedyre `lm-train!` som itererer gjennom alle ordpar i alle setninger fra Brown-korpuset og teller opp bruksfrekvensene i en 'språkmodell'. Test resultatene ved å slå opp ordpar som også kan telles manuelt i tekstfilen '`brown.txt`', for eksempel har $\langle \textit{jury}, \textit{said} \rangle$ en frekvens på 3.

(c) For å gå fra den initiale 'språkmodellen' (dvs. tabellen med råfrekvenser) til en 'ekte' språkmodell med betingete sannsynligheter trenger vi en ny prosedyre som regner ut $P(w_i|w_{i-1})$ for alle ordpar utfra formelen over; i tillegg til ordparets frekvens vil denne utregningen altså kreve kunskap om det totale antall forekomster av enkeltordet w_{i-1}:

- `lm-estimate` *lm* — returnerer en 'ny' språkmodell der de betingete sannsynlighetene er regnet ut fra frekvenstallene i *lm*. Det er fullt mulig å skrive en destruktiv `lm-estimate!` som bare gjør om på verdiene lagret i *lm*, fra å være frekvenser til sannsynligheter (isåfall vær obs på å gjøre ting i riktig rekkefølge).

Strengt tatt kan man tenke seg å skille mellom to abstrakte datatyper: (i) tabellen med råfrekvensene, som blant annet støtter `lm-record-bigram!` og `lm-estimate`, vs. (ii) den 'ekte' språkmodellen som inneholder sannsynlighetene og støtter `lm-score` (se under) men ikke telling av nye bigrammer eller estimering. Dette er et fritt designvalg, men selv om man ser på de to som konseptuelt forskellige må de gjerne ha mye felles 'innmat', dvs. indeksstrukturen, og overgangen fra (i) til (ii) kan muligens implementeres destruktivt likevel.

Test gjerne at du får riktige resultater *før* og *etter* estimeringen. Hva blir bigramsannsynligheten $P(\textit{said}|\textit{jury})$, for eksempel?

(d) Nå skriv en prosedyre for å bruke en språkmodell til å regne ut $P(s)$:

- `lm-score` *lm sentence* — returnerer produktet av bigramsannsynlighetene for alle ordpar i *sentence*, dvs. $P(\textit{sentence})$.

I filen `test.txt` finnes det en liste over setningsvarianter som alle har samme betydning (men forskjellig ordstilling). Hvilken variant er mest sannsynlig ifølge Brown-språkmodellen, og hvilken er minst sannsynlig?

(e) Jo større treningsmateriale som brukes jo bedre kan man forvente at en språkmodell speiler den språklige 'virkeligheten', dvs. preferanser som gjelder på tvers av språkbrukere og sjangere. I filen '`wsj.txt`' får dere en litt større mengde tekst som muligens kan by på effisiensutfordringer for en naïv implementering av språkmodellens datastrukturer. Med mindre du har gjort dette allerede fra begynnelsen, bruk binære trær for å optimere tidskompleksitet i `lm`-prosedyrene. Tren sannsynlighetene fra bare '`wsj.txt`' og fra begge korpusfilene sammen; blir det synlige forskjeller i preferansene blant setningene fra '`test.txt`'?

Lykke til, og god koding!