Innlevering 3a, IN2040 høst 2021

- Innlevering 3a er del én av den siste obligatoriske oppgaven i IN2040. Man må ha minst 12 poeng tilsammen for 3a + 3b, og man kan få opptil 10 poeng på hver innlevering.
- Svarene skal leveres i Devilry *innen fredag, 5. november, kl. 23:59*. Ta også med relevante kjøringseksempler.
- Utover gruppetimene kan spørsmål som vanlig postes på GitHub:
  `https://github.uio.no/IN2040/h21/issues/`
- **Gruppearbeid:** Som for oblig 2 kan oppgavene i oblig 3 også løses gruppevis (2–3 studenter sammen). Merk at det ikke er anledning til å bytte grupper mellom a- og b-leveringen (siden dette skaper problemer for poengtildelingen i Devilry). Det holder at én på hver gruppe leverer, men la første linje i besvarelsen gjøre det klart hvem andre som er med på gruppa. Gruppen 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`
- Dersom du ikke ønsker å jobbe i gruppe og heller vil levere individuelt er det også helt greit.
- **Prekode:** Fila *prekode3a.scm* inneholder en del hjelpeprosedyrer som dere kan bruke om dere vil; se oblig-mappen i GitHub-Repoet. Koden inkluderer datatyper for både tabeller og strømmer. I tillegg inneholder den 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 liggere i samme mappe).
- Hvis kildekoden leveres som en annen filtype (altså ikke som *.scm*-fil) vil dette anses som feil og det vil bli trukket 2 poeng.

1 Prosedyrer for bedre prosedyrer

I denne oppgaven skal vi implementere en teknikk som kalles *memoisering* (*memoization*) (som diskutert i forelesningene 8 og 9, 14. og 21. oktober). Dette er et enkelt eksempel på såkalt *dynamisk programmering* der vi lar programmet huske (*cache*) tidligere utførte beregninger. Dette kan gi store besparelser for prosedyrer som utfører ressurskrevende beregninger: Dersom en memoisert prosedyre kalles med samme argument flere ganger vil beregningen kun bli utført én gang, ved første kall, mens senere kall bare slår opp returverdien i en tabell der vi har lagret det tidligere resultatet.

Vi skal her skrive en prosedyre `mem` som tar en prosedyre som argument og returnerer en ny memoisert versjon av prosedyren (se oppgave *a* under). Oppgave 3.27 i SICP gir et eksempel på en memoiseringsprosedyre, og man kan godt bruke dette til inspirasjon. Men her ønsker vi å definere en forbedret og mer fleksibel variant som fungerer litt anderledes. I tillegg til å kunne generere en memoisert versjon av en gitt prosedyre ønsker vi også å kunne 'av-memoisere' og gjenoprette bindingen til den opprinnelige prosedyren (oppgave *b* under). Vi skal også støtte memoisering av prosedyrer som tar et vilkårlig antall argumenter. Kalleksempler følger under.

Vi har allerede på plass funksjonalitet for oppslag og innsetting i tabeller: '*prekode3a.scm*' inkluderer tabell-prosedyrene fra seksjon 3.3.3 i SICP (og som gjennomgått i 8. uke). Fila inneholder også definisjoner av `fib`

og `test-proc` som kan brukes som testprosedyrer: Utover å bare beregne returverdiene sine inneholder disse prosedyrene også noen print-kommandoer så det blir lettere å sjekke at memoiseringen fungerer som den skal. (Det kan være nyttig å ta en titt på disse prosedyrene nå før du leser videre.)

I oppgave *a* og *b* under skal vi definere en prosedyre `mem` som fungerer som følger:

```
? (set! fib (mem 'memoize fib))
? (fib 3)
computing fib of 3
computing fib of 2
computing fib of 1
computing fib of 0
→ 2
? (fib 3)
→ 2
? (fib 2)
→ 1
? (fib 4)
computing fib of 4
→ 3
? (set! fib (mem 'unmemoize fib))
? (fib 3)
computing fib of 3
computing fib of 2
computing fib of 1
computing fib of 0
computing fib of 1
→ 2
```

Vi ser at den nye memoiserte versjonen av `fib` kun beregner Fibonacci-funksjonen for et gitt tall én gang. Ved gjentatte kall med samme argument vil vi at returverdien bare slås opp i en tabell. Merk: Når `mem` kalles med beskjeden `'unmemoize` som første argument så returneres den opprinnelige prosedyren (nest siste kall over). Merk også at `mem` skal være generell nok til å kunne generere memoiserte versjoner av prosedyrer som tar et vilkårlig antall argumenter. Dette kan sjekkes med f.eks `test-proc`:

```
? (set! test-proc (mem 'memoize test-proc))
? (test-proc)
computing test-proc of ()
→ 0
? (test-proc)
→ 0
? (test-proc 40 41 42 43 44)
computing test-proc of (40 41 42 43 44)
computing test-proc of (41 42 43 44)
computing test-proc of (42 43 44)
computing test-proc of (43 44)
computing test-proc of (44)
→ 10
? (test-proc 40 41 42 43 44)
→ 10
? (test-proc 42 43 44)
→ 5
```

(a) Skriv prosedyren `mem` som gitt beskjeden `'memoize` og en prosedyre returnerer en memoisert versjon av denne.

(b) Denne delen er litt mer vrien: Utvid prosedyren `mem` til å også støtte beskjeden `'unmemoize` slik at vi kan gjenopprette den opprinnelige ikke-memoiserte versjonen av en prosedyre. (Merk at det holder å levere én prosedyre `mem` for både *a* og *b* dersom du får til begge. Merk også at vi ved av-memoiseringen ikke bryr oss om å slette eventuelt lagrete resultater fra minnet.)

(c) Sammenlikn måten vi bruker `mem` på over med hvordan vi gjør det her:

```
? (define mem-fib (mem 'memoize fib))
? (mem-fib 3)
computing fib of 3
computing fib of 2
computing fib of 1
computing fib of 0
computing fib of 1
→ 2
? (mem-fib 3)
→ 2
? (mem-fib 2)
computing fib of 2
computing fib of 1
computing fib of 0
→ 1
```

Det kan se ut til at `mem-fib` ikke oppfører seg helt som vi vil! Forklar hva som er problemet her. Tips: 'Sidesporet' vi snakket om på foilene 23–26 fra uke nr. 8 kan være til hjelp her.

2 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) Skriv en prosedyre `stream-take` som gitt et heltall `n` og en strøm `stream` som argumenter returnerer en ny strøm av de `n` første elementene i `stream`. Kalleksempler:

```
? (define foo (stream-take 10 nats))
? foo
→ (1 . #<promise>)
? (show-stream foo 5)
⇝ 1 2 3 4 5 ...
? (show-stream foo 20)
⇝ 1 2 3 4 5 6 7 8 9 10
? (show-stream (stream-take 15 nats) 10)
⇝ 1 2 3 4 5 6 7 8 9 10 ...
```

(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) Fullfør følgende definisjon av en prosedyre som korrekt genererer en ny strøm uten duplikatene i input-strømmen. Merk at prosedyren er ment å inkludere et kall på `stream-filter` som er gitt i '*prekode3a.scm*' og er definert på samme måte som i SICP seksjon 3.5.1: Gitt et predikat og en strøm returnerer `stream-filter` en ny strøm som består av de elementene fra input-strømmen som oppfyller predikatet (dvs. returnerer sant).

```
(define (remove-duplicates stream)
  (if (stream-null? stream)
      the-empty-stream
      (cons-stream <FIRST-ELEMENT>
                   (remove-duplicates
                     <CALL-ON-STREAM-FILTER>))))
```

Lykke til og god koding!