Innlevering 2b i INF2810, vår 2017

- Dette er del to av den andre obligatoriske oppgaven i INF2810. Man kan oppnå 10 poeng for oppgavene i 2b, og man må ha minst 12 poeng tilsammen for 2a + 2b for å få godkjent.
- Svarene skal leveres via Devilry *innen fredag 6. april kl 15:00*.
- I enkelte oppgaver ber vi dere tegne diagrammer: Husk at disse også må vedlegges besvarelsen (f.eks som bilder av tegninger med penn-og-papir), i tillegg til kildekoden (som leveres som en *.scm*-fil).
- Husk å kommentere koden (med `;;`) så det blir lettere for de som skal rette å skjønne hvordan dere har tenkt. Ta også med relevante kjøringseksempler (ev. med returverdiene kommentert ut).
- For dem som levererer gruppeoppgaver: 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.
- Forelesningene fra 2/3, 9/3, og 16/3 er de mest relevante her. Utover gruppetimene kan spørsmål som vanlig postes på Piazza.
- **Prekode:** Fila *prekode2b.scm* inneholder noen hjelpeprosedyrer som dere kan bruke i oppgave 4 under:
  `uio.no/studier/emner/matnat/ifi/INF2810/v17/oppgaver/prekode2b.scm`
  Koden inkluderer tabellprosedyrene fra seksjon 3.3.3 i SICP samt noen testprosedyrer. Dette kan lastes inn med `(load "prekode2b.scm")` øverst i kildefilen din (gitt at filene liggere i samme mappe). Hvis du heller vil lage dine egne implementasjoner av noe fra *prekode2b.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).

1 Innkapsling, lokal tilstand og omgivelsesmodellen

(a) Skriv en prosedyre `make-counter` som returnerer en ny prosedyre som bruker innkapsling for å holde rede på hvor mange ganger den har blitt kalt. Den returnerte prosedyren skal ha en privat variabel `count` som initialiseres til 0 og så økes med 1 (destruktivt) hver gang vi kaller den. Som returverdi gis den oppdaterte verdien til `count`. Eksempel på interaksjon:

```
? (define count 42)
? (define c1 (make-counter))
? (define c2 (make-counter))
? (c1) → 1
? (c1) → 2
? (c1) → 3
? count → 42
? (c2) → 1
```

(b) Tegn et omgivelsesdiagram som viser alle rammene og bindingene som er gjeldende idet vi evaluerer det siste uttrykket i interaksjonen over, altså kallet på `c2`. Merk at vi her er interessert i omgivelsen som er gjeldene mens kallet utføres, ikke etter at det er ferdig og har returnert.

2 Innkapsling, lokal tilstand og *message passing*

(a) I denne oppgaven skal vi implementere en abstrakt datatype for å representere stakker (*stacks*). En stakk er en 'beholder' der vi kan legge til elementer (*push*) eller fjerne elementer (*pop*). Fjerning og innsetting av elementer i en stakk er basert på en strategi som kan beskrives som *sist-inn-først-ut*, eller *last-in-first-out* (LIFO): Det siste elementet som er lagt til er det første som vil bli fjernet. Den klassiske metaforen her er en stabel med tallerkener: Vi kan kun forsyne oss fra toppen, og den siste tallerkenen som er lagt i stabelen er den første som kan tas ut igjen.

Vi skal implementere stakker som prosedyreobjekter der elementene på stakken er representert som en liste bundet til en lokal variabel (ved å bruke *innkapsling*). Vi kommuniserer med prosedyreobjektene ved å bruke såkalt *message passing*. Det er tre beskjeder vi ønsker å kunne gi (se eksempler på kall under):

- `'push!`, sammen med et vilkårlig antall elementer som (destruktivt) skal legges til i stakken.
- `'pop!` (uten andre argumenter), for å (destruktivt) fjerne det siste / 'øverste' elementet i stakken.
- `'stack` (uten andre argumenter), for å returnere lista av elementer som er på stakken.

Skriv en prosedyre `make-stack` som returnerer en stakk slik at vi kan ha f.eks. følgende interaksjon:

```
? (define s1 (make-stack (list 'foo 'bar)))
? (define s2 (make-stack '()))
? (s1 'pop!)
? (s1 'stack) → (bar)
? (s2 'pop!)     ;; popper en tom stack
? (s2 'push! 1 2 3 4)
? (s2 'stack) → (4 3 2 1)
? (s1 'push! 'bah)
? (s1 'push! 'zap 'zip 'baz)
? (s1 'stack) → (baz zip zap bah bar)
```

Det bør være mulig å 'poppe' en tom stakk uten å få feilmelding (vi lar den tomme stakken forbli uendret).

(b) Videre ønsker vi å gjøre grensesnittet mot stakk-objektene mer generelt slik at vi 'abstraherer bort' det faktum at de her tilfeldigvis er implementert som prosedyrer. Definer prosedyrene `push!`, `pop!` og `stack` som alle tar et stakk-objekt som argument. (I tilfellet `push!` skal den i tillegg godta et vilkårlig antall elementer som skal settes inn.) Gitt stakk-objektet `s1` (i samme tilstand som vi etterlot det i 2a-eksemplet over) ønsker vi å kunne ha f.eks. følgende interaksjon:

```
? (pop! s1)
? (stack s1) → (zip zap bah bar)
? (push! s1 'foo 'faa)
? (stack s1) → (faa foo zip zap bah bar)
```

3 Strukturdeling og sirkulære lister

- I denne oppgaven skal vi jobbe med sykliske lister og strukturdeling. La oss anta følgende REPL-interaksjon:

```
? (define bar (list 'a 'b 'c 'd 'e))
? (set-cdr! (cdddr bar) (cdr bar))
? (list-ref bar 0) → a
? (list-ref bar 3) → d
? (list-ref bar 4) → b
? (list-ref bar 5) → c
```

(a) Tegn to boks-og-peker-diagrammer som viser strukturen til `bar` både etter kallet på `define` og etter `set-cdr!` over. Forklar kort hvorfor vi får verdiene vi gjør ved kallene på `list-ref`.

(b) Anta så at vi har følgende interaksjon:

```
? (define bah (list 'bring 'a 'towel))
? (set-car! bah (cdr bah))
? bah → ((a towel) a towel)
? (set-car! (car bah) 42)
? bah → ((42 towel) 42 towel)
```

Tegn boks-og-peker-diagram som viser strukturen til `bah` før og etter det første kallet på `set-car!`. Forklar kort hvorfor `bah` evaluerer til verdien den gjør etter det siste kallet på `set-car!`.

(c) Strengt tatt er sirkulære lister egentlig ikke ekte lister. For `bar` og `bah` (slik de er gitt ved REPL-interaksjonen i oppgavene over) vil vi få følgende resultat hvis vi kaller predikatet `list?` – forklar kort hvorfor.

```
? (list? bar) → #f
? (list? bah) → #t
```

4 Dynamisk programmering: memoisering

I denne oppgaven skal vi implementere en teknikk som kalles memoisering (*memoization*). 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 annerledes. I tillegg til å kunne generere en memoisert versjon av en gitt prosedyre ønsker vi også å kunne 'av-memoisere' og gjenopprette bindingen til den opprinnelige prosedyren (oppgave *b* under). Vi skal også støtte memoisering av prosedyrer som tar et vilkårlig antall argumenter. Kalleksempler følger under. Merk at forelesningen fra 16/3 vil være spesielt relevant for hele denne oppgaven.

Vi har allerede på plass funksjonalitet for oppslag og innsetting i tabeller: '*prekode2b.scm*' inkluderer tabell-prosedyrene fra seksjon 3.3.3 i SICP (som gjennomgått på forelesning 16/3). 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 17–20 på 9. forelesning (16. mars) kan være til hjelp her.

Lykke til, og god koding!