

Innlevering 2b

IN2040 (Høst 2021)

- Dette er del to av den andre obligatoriske oppgaven i IN2040. 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 22. oktober, kl. 23:59*.
- 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).
- Hvis kildekoden leveres som en annen filtype (altså ikke som `.scm`-fil) vil dette anses som feil og det vil bli trukket 2 poeng.
- 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 leverer 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. (Som for 2a holder det at én på hver gruppe leverer, men la første linje i besvarelsen gjøre det klart hvem andre som er med på gruppa og sørg for at gruppen er registrert i Devilry.)
- Forelesningene fra uke 6 (30. september), 7 (7. oktober) og til en viss grad uke 8 (14. oktober) er de mest relevante her. Utover gruppetimene kan spørsmål som vanlig postes på GitHub-forumet.

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) \rightarrow (bar)

```
? (s2 'pop!)      ;; popper en tom stack
```

```
? (s2 'push! 1 2 3 4)
```

? (s2 'stack) \rightarrow (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! (cddddr 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 boks-og-peker-diagram som viser strukturen til `bar` 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) Strukturen `bar` over er et eksempel på en sirkulær (*cyclic*) liste. Med sirkulære lister mener vi her strukturer hvor et `cdr`-felt peker tilbake til en tidligere `cons`-celle i lista. (Merk at lister kan ha elementer som deler struktur uten å nødvendigvis være sirkulære gitt denne definisjonen.) Når vi jobber med sirkulære lister må vi dermed passe oss så det ikke oppstår uendelige løkker der vi rekurserer over lister med `cdr`. Hvis vi f.eks. forsøker å kalle `length` på `bar` vil den ikke terminere. Skriv et predikat `cycle?` som tester om en listestruktur er syklisk.

```
? (cycle? '(hey ho)) → #f
```

```
? (cycle? '(la la la)) → #f
```

```
? (cycle? bah) → #f
```

```
? (cycle? bar) → #t
```

For ordens skyld: Vi bryr oss ikke her om “dypere” sirkularitet — f.eks. i tilfeller med lister-av-lister bryr vi oss ikke om hvorvidt et liste-element selv inneholder en sirkulær liste.

Valgfritt: De som kan tenke seg en ekstra utfordring kan her prøve å skrive to versjoner: en som har lineær minnebruk og en som har konstant minnebruk (men lineær tidsbruk).

- (d) 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
```

Lykke til, og god koding!