UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i: INF2810
Eksamensdag: Fredag 5. juni 2015
Tid for eksamen: 14:30 (4 timer)
Oppgavesettet er på 4 sider (ikke medregnet denne siden)
Vedlegg: Ingen
Tillatte hjelpemidler: Ingen

Kontroller at oppgavesettet er komplett
før du begynner å besvare spørsmålene.

Eksamen, INF2810, vår 2015

Bakgrunn

- Vi anbefaler å lese gjennom hele oppgaveteksten før du begynner (4 sider). Hvis du føler du savner informasjon for å løse en oppgave, gjør dine egne antakelser og redegjør for dem.
- Alle steder der det bes om kode forventes i utgangspunktet Scheme (i henhold til R5RS slik vi har brukt hele semesteret). Dersom du for eksempel ikke husker spesifikk Scheme-syntaks eller visse prosedyrenavn er det bedre om du skriver pseudokode med kommentarer enn ingenting.
- Som i forelesningnotatene brukes enkelte steder "→" for indikere verdien et uttrykk evaluerer til.
- De mulige poengene angitt for hver seksjon summerer til 100.

1 Par, lister og rekursjon (25 poeng)

(a) Se nøye på sekvensen av uttrykk som kalles under. Tegn et boks-og-peker-diagram som viser strukturen som `foo` er bundet til etter at det siste uttrykket er evaluert.

```
(define foo (list "bring" "towels" "!"))

foo → ("bring" "towels" "!")

(set-car! foo foo)

(set-cdr! (cdr foo) 42)
```

(b) Vi antar her at `list?` ikke er innebygget. Implementer predikatet `list?` selv. Det skal ta ett argument og returnere `#t` dersom det er en liste og `#f` ellers. For å få full uttelling skal du hverken bruke `if` eller `cond`. Du trenger ikke ta høyde for at argumentet kan være en uendelig syklisk struktur.

(c) Hva synes du predikatet ditt `list?` burde returnere dersom argumentet var `foo` (altså etter at hele sekvensen i 1a er evaluert)? Begrunn svaret ditt med én setning.

(d) I denne oppgaven skal vi jobbe med nøstede lister, altså lister som kan inneholde andre (også nøstede) lister. Skriv en rekursiv prosedyre `deep-map` som tar to argumenter: en prosedyre `proc` og en (muligens nøstet) liste `nested`. Den skal returnere en ny nøstet liste med `proc` anvendt på hvert atomære element i `nested`. Løs oppgaven uten å bruke den innebygde `map`. Kalleksempel:

```
(define nested '(((1) 2) 3 (4 (5 6))))

(deep-map (lambda (x) (* x 10))
          nested)

→ (((10) 20) 30 (40 (50 60)))
```

(e) Med utgangspunkt i kalleksemplet ovenfor, hva slags prosess gir prosedyren `deep-map` opphav til? Hva kan du si om tidskompleksiteten (vekst i antall trinn)?

2 Paradigmer og idiomer (25 poeng)

I denne oppgaven skal du skrive noen forskjellige versjoner av en enkel prosedyre `replace`. Det anbefales å lese igjennom alle deloppgavene før du setter i gang. Argumentene skal være to symboler `x` og `y` og en liste av symboler `seq`. Returverdien skal være en liste der hver forekomst av `x` er erstattet med `y`, mens resten av elementene er de samme. Kalleksempel:

```
(define foo '(a b b a))

(replace 'a 'c foo) → (c b b c)
```

(a) Skriv en rent funksjonell versjon av `replace` basert på halerekursjon.

(b) Skriv en rent funksjonell versjon av `replace` basert på vanlig rekursjon.

(c) Skriv en rent funksjonell versjon av `replace` basert på høyereordens sekvensoperasjoner (benytt deg gjerne av innebygde prosedyrer her).

(d) Skriv en destruktiv versjon av `replace` som modifiserer listeargumentet sitt (og returnerer den modifiserte listen).

(e) Skriv en strømversjon av løsningen din for deloppgave (b) over: Både argumentet og returverdien skal være en strøm av symboler. Du kan her anta at hele grensesnittet for strømmer slik vi har brukt det i kurset er tilgjengelig.

(f) Med utgangspunkt i kalleksempelet på sekvensen `foo` over, hvor mange `cons`-operasjoner vil utløses av å kalle de respektive `replace`-prosedyrene dine fra (b), (d) og (e)? For (e) antar du at sekvensen er en strøm i stedet for en liste, men med de samme fire elementene.

3 Funksjonelle prosedyrer (12 poeng)

(a) Forklar med kun et par setninger hva du mener skiller en rent funksjonell prosedyre fra en ikke-funksjonell prosedyre.

(b) Den innebygde konstruksjonen `begin` lar oss evaluere en sekvens av uttrykk. Verdien til det siste uttrykket i sekvensen blir også returverdien til `begin`-uttrykket selv. Denne konstruksjonen kan være nyttig i forbindelse med f.eks. `if`-uttrykk, som jo i utgangspunktet forventer at hvert argument angir ett enkelt uttrykk, se eksempelkall under.

Ta stilling til følgende påstand: Så lenge vi holder oss til ren funksjonell kode har vi ingen bruk for `begin`. Foklar med noen få setninger hvorfor du mener dette stemmer eller ikke stemmer.

```
(if (some-test)
 (begin
   (do-something-1)
   (do-something-2))
 (do-something-3))
```

(c) Forklar med én setning hva vi mener med begrepet *høyereordens prosedyrer*.

4 Innkapsling og omgivelser (16 poeng)

(a) Vi har sett at vi kan implementere en abstrakt datatype for par / cons-celler basert på prosedyrer. I denne oppgaven antar vi at grensesnittet er definert som følger (vi later altså som at par ikke er innebygget i språket):

```
(define (cons x y)
  (lambda (message)
    (cond ((eq? message 'car) x)
          ((eq? message 'cdr) y))))

(define (car p)
  (p 'car))

(define (cdr p)
  (p 'cdr))
```

Her er noen kalleksempler, gitt definisjonene over:

```
(define foo (cons 1 (cons 2 3)))

(car foo) → 1

(car (cdr foo)) → 2

(cdr (cdr foo)) → 3
```

Tegn et omgivelsesdiagram som viser verdien som `foo` er bundet til, gitt interaksjonen vist ovenfor. (Du trenger ikke vise bindingene til andre variabler i den globale omgivelsen slik som `cons`, `car`, osv.)

(b) En ting som mangler i grensesnittet over er støtte for de destruktive prosedyrene `set-car!` og `set-cdr!` som lar oss endre hvilke verdier som lagres i par. Legg til definisjoner for disse i grensesnittet vårt. Du vil også måtte utvide definisjonen av `cons` over. I implementasjonen din har du har lov til å bruke den innebygde `set!` men ikke de to mutatorene vi selv skal definere. Vi ønsker å kunne ha f.eks. følgende interaksjon:

```
(car foo) → 1

(set-car! foo 'a)

(car foo) → a

(set-cdr! (cdr foo) 'c)

(cdr (cdr foo)) → c
```

5 Mer innkapsling (10 poeng)

Denne oppgaven handler om å representere køer. En kø defineres her med en maksimal lengde n. Dersom en kø er n elementer lang når det legges til et nytt element, så vil et annet element automatisk fjernes. Køene skal fungere etter prinsippet først-inn-først-ut: Nye elementer legges til bakerst og om nødvendig slettes det eldste elementet fra begynnelsen av listen.

Definer en prosedyre `make-queue` som tar ett argument, et heltall `n`, og returnerer et prosedyreobjekt som ved hjelp av innkapsling lar oss huske opptil `n` elementer om gangen i en liste. Anta at `n` > 1. Kalleksempler:

```
(define foo (make-queue 3))

(define bar (make-queue 2))

bar → #<procedure>

(bar 'a) → (a)

(bar 'b) → (a b)

(bar 'c) → (b c)

(bar 'd) → (c d)

(foo 1) → (1)

(foo 2) → (1 2)

(foo 3) → (1 2 3)

(foo 4) → (2 3 4)
```

Vi ser altså at `make-queue` returnerer ett-arguments prosedyrer som vi her binder til `foo` og `bar`. Hvert kall på `foo` og `bar` legger til et nytt element og returnerer en oppdatert liste av elementer. Køen skal ikke bli lengre enn hva vi har spesifisert og om nødvendig slettes det eldste elementet for å gjøre plass til et nytt.

Tidskompleksiteten ved lagring og sletting av elementer bør være konstant og uavhengig av størrelsen på n.

6 Strømmer og evaluereringsstrategier (12 poeng)

(a) Vi antar her at hele grensesnittet for å jobbe med strømmer slik vi har brukt det i kurset er forhåndsdefinert og tilgjengelig. I tillegg definerer vi en prosedyre `add-inf-streams` som følger.

```
(define (add-inf-streams s1 s2)
  (cons-stream (+ (stream-car s1) (stream-car s2))
               (add-inf-streams (stream-cdr s1) (stream-cdr s2))))
```

Den rekursive prosedyren `add-inf-streams` forventer to uendelige strømmer som argument og returnerer en ny strøm med summene av de parvise elementene fra strømargumentene. Prosedyren er veldefinert slik den står (i den forstand at den fungerer som intendert hvis vi kaller den). Sammenliknet med de fleste andre rekursive prosedyredefinisjoner vi har sett på så er det likevel et vanlig element som er fraværende i denne definisjonen. Hva er det som mangler, og hvorfor er den likevel veldefinert?

(b) La oss si at vi erstatter `cons-stream` med `cons` i `add-inf-streams` over, men ellers lar vi definisjonen stå uendret. Hva vil nå bli resultatet av å kalle `add-inf-streams`? Begrunn svaret ditt.

(c) Se nøye på uttrykkene i sekvensen under. Hva blir resultatet når vi evaluerer det første uttrykket? Hva blir resultatet når vi evaluerer det andre uttrykket? Forklar kort hvorfor i begge tilfeller.

```
(define ones (cons-stream 1 ones))

(define integers (add-inf-streams ones integers))
```