Prøveeksamen i INF2810

Bakgrunn

- Vi anbefaler å lese gjennom hele oppgaveteksten før du begynner. 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), men dersom du av en eller annen grunn står fast noe sted og for eksempel ikke husker spesifikk Scheme-syntaks eller prosedyrenavn er det bedre om du skriver pseudokode med kommentarer enn ingenting.

1 Grunnleggende (6 poeng)

Gitt følgende sekvens av uttrykk, hva er verdien til `foo` etter at *siste* uttrykk har blitt evaluert? Tegn også boks-og-peker-diagram som viser strukturen.

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

? (define bar foo)

? (set! foo '(c d))

? (set-car! foo bar)

? (set-car! (cdr foo) (car foo))

? (set-cdr! bar (list 7))
```

2 Mysterium (6 poeng)

Forklar med én setning hva følgende prosedyre gjør. Gi også et konkret eksempel på et kall og tilsvarende returverdi som viser hvordan prosedyren kan brukes (du må altså selv velge hva argumentene skal være).

```
(define (x y z)
  (or (null? z)
      (and (y (car z))
           (x y (cdr z)))))
```

3 Rekursjon (8 poeng)

Prosedyren `avg` under forventer ett eller flere tall som argumenter og returnerer gjennomsnittet:

```
? (define (avg first . rest)
    (let ((arguments (cons first rest)))
      (/ (apply + arguments)
         (length arguments))))
```

```
? (avg 1 2 3) → 2

? (avg 1 1 2 2 3 3) → 2
```

Definer en ny rekursiv variant av `avg` (uten bruk av `apply` og `length`). Den kan gjerne benytte en intern hjelpeprosedyre om du vil.

4 Omgivelser (10 poeng)

Her skal vi jobbe med omgivelsesmodellen for evaluering. Tegn et diagram som viser gjeldende omgivelse idet kallet `(b sum)` evalueres i det siste uttrykket i følgende sekvens (returverdiene vises ikke):

```
? (define sum 100)

? (define (make-acc sum)
    (lambda (x)
      (set! sum (+ sum x))
       sum))

? (define a (make-acc sum))

? (define b (make-acc 200))

? (a 10)

? (let ((sum 30))
    (b sum))
```

5 Høyereordens prosedyrer (14 poeng)

La oss tenke oss en høyereordens prosedyre `transform-if` som tar tre argumenter: `test`, `trans` og `seq`. Vi forventer at `seq` er en liste med elementer, og `transform-if` skal returnere en ny liste der vi har anvendt prosedyren `trans` på hvert element som tester sant for predikatet `test`. Elementer som ikke oppfyller testen inkluderes også i lista men forblir uendret. Eksempel på kall:

```
? (define foo '(1 2 3 4 5 6))

? (transform-if odd? (lambda (x) (+ x 1)) foo) → (2 2 4 4 6 6)
```

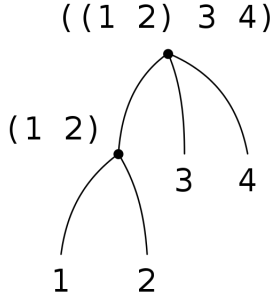
Skriv to varianter av denne prosedyren: (a) én som er rent funksjonell (ikke bruker destruktive operasjoner) og genererer en ny liste; og (b) en annen (kall den `transform-if!`) som bruker destruktive operasjoner og modifiserer sitt listeargument `seq` . Diskuter også kort (noen få setninger) forskjellen på de to variantene med tanke på minnebruk.

6 Evalueringsstrategier (16 poeng)

Forklar kort hovedforskjellen mellom såkalt *eager* eller *applicative-order evaluation* på den ene siden og *lazy* eller *normal-order evaluation* på den andre. Diskuter også kort (maksimalt en halv side) og generelt hvor vi har sett disse to ulike strategiene i bruk (eksemplene kan være basert på funksjonalitet som er innebygget i Scheme eller kode vi ellers har jobbet med i forelesninger eller oppgaver).

7 Dataabstraksjon og strømmer (20 poeng)

Vi har sett hvordan trær kan representeres som lister av lister. I dette eksemplet skal vi tenke at hvert element i en liste danner en gren; elementer som selv er lister er subtrær; og løvnodene i treet er de atomære elementene som ikke er lister. Som et eksempel vil listen `'((1 2) 3 4)` represente treet vi ser tegnet under. Her tenker vi altså at vi har verdier kun på løvnodene.



En vanlig operasjon man ønsker å definere for trær er å lage en flat liste av alle nodeverdiene. For eksempel;

```
? (fringe '((1 2) 3 4)) → (1 2 3 4)

? (fringe '((1 (2)) 3 (4))) → (1 2 3 4)
```

En prosedyre fringe for å gjøre dette er definert under;

```
(define (fringe tree)
  (cond ((null? tree) '())
        ((pair? (car tree))
         (append (fringe (car tree))
                 (fringe (cdr tree))))
        (else (cons (car tree)
                    (fringe (cdr tree))))))
```

(a) Skriv et predikat `same-fringe?` som basert på `fringe` over rekursivt sjekker om to trær har like løvnoder. Prosedyren skal ta tre argumenter; de to trærne som skal sammenliknes, i tillegg til et predikat som skal brukes til å sammenlikne verdiene i listene av løvnoder. Det siste parameteret er med for at prosedyren ikke skal gjøre noen spesielle antakelser om hva slags datatyper vi har i løvnodene. I trærne over har vi numeriske verdier så da kan vi sammenlikne med =, og noen eksempler på kall kan være følgende;

```
? (same-fringe? '((1 2) 3 4)
                '((1 (2)) 3 (4))
                =)
→ #t

? (same-fringe? '((1 2) 3 4)
                '((1 7) 3 4)
                =)

→ #f
```

(b) Du skal nå definere nye versjoner av prosedyrene over som skal benytte seg av *strømmer*, vi kaller dem `fringe-stream` og `same-fringe-stream?`. Merk at `fringe-stream` skal returnere en strøm av løvnoder for et gitt tre, men vi tenker at trærne selv fortsatt er gitt som vanlige lister, akkurat som i kalleksemplene over. Predikatet `same-fringe-stream?` skal ta akkurat samme argumenter som `same-fringe?`

over men skal benytte seg av den nye strøm-prosedyren `fringe-stream`. Du kan anta at vi allerede har et grensesnitt for å jobbe med strømmer som inkluderer operasjonene vi har brukt på forelesningene: `cons-stream`, `stream-car`, `stream-cdr`, `stream-null?`, og (konstanten) `the-empty-stream`. I tillegg kan vi anta at vi har `stream-append`, definert som følger;

```
(define (stream-append s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (stream-append (stream-cdr s1) s2))))
```

Du kan også definere flere egne strømoperasjoner hvis du vil.

(c) Gitt implementasjonen din over, hvor mange cons-celler genreres i hvert av de to kallene under? Forklar også kort hvorfor.

```
? (fringe-stream '(1 ((2) 3 4)))

? (fringe-stream '((1 2) 3 4))
```

(d) I hvilke tilfeller vil det være en fordel å bruke den siste strømbaserte varianten fremfor den første? Gi en kort begrunnelse.

8 Kallstatistikk (20 poeng)

I denne oppgaven skal vi implementere støtte for såkalt *call counting*, dvs. statistikk over hvor mange ganger en prosedyre blir kallt opp. Vi vil at dette kan anvendes på en hvilken som helst brukerdefinert prosedyre, og uten å benytte seg av global tilstand. Funksjonaliteten skal være som følgende:

```
? (define original fringe)
? (set! fringe (monitor fringe))
? (fringe '((1 2) 3)) → (1 2 3)
? (fringe 'count) → 6
? (fringe '((1 2) 3)) → (1 2 3)
? (fringe 'count) → 12
? (fringe 'zero)
? (fringe '((1 2) 3)) → (1 2 3)
? (fringe 'count) → 6
? (set! fringe (fringe 'reset))
? (eq? fringe original) → #t
```

I eksemplene over blir vår opprinnelige prosedyre `fringe` redefinert med et utvidet grensesnitt: I tillegg til kallene vi har sett tidligere kan den nye `fringe` også kalles med bare et *message*-argument, og dette brukes til å manipulere kall-telleren i stedet for å kalle den opprinnelige prosedyren. Beksjeden `count` returnerer tellerens nåværende verdi, beskjeden `zero` nullstiller den, og `reset` frigjør prosedyren fra kallstatistikk, dvs. returnerer den opprinnelige prosedyren. Selv om `fringe` i dette eksemplet bare tar ett argument, så bør mekanismen for kallstatistikk ikke gjøre noen antakelser om antall argumenter som den opprinnelige prosedyren måtte ta.