# UNIVERSITETET I OSLO

## Det matematisk-naturvitenskapelige fakultet

| | |
|---|---|
| **Eksamen i:** | **INF2810** |
| **Eksamensdag:** | **5. juni, 2014** |
| **Tid for eksamen:** | **14:30 (4 timer)** |
| **Oppgavesettet er på 4 sider.** | |
| **Vedlegg:** | **Ingen** |
| **Tillatte hjelpemidler:** | **Ingen** |

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

# Eksamen i INF2810, vår 2014

## 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), 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.
- Som i forelesningnotatene brukes enkelte steder "→" for indikere verdien et uttrykk evaluerer til.

## 1 Listestrukturer (16 poeng)

(a) Gitt definisjonene under, hva er verdiene til `z1` og `z2`? Tegn også boks-og-peker-diagrammer som viser de underliggende strukturene.

```
(define z1
  (let ((foo (list 'a 'b)))
    (cons foo foo)))

(define z2
  (cons (list 'a 'b) (list 'a 'b)))
```

(b) Igjen, gitt definisjonene over, forklar hva som blir effekten av følgende kall (det er også greit å vise dette ved å tegne boks-og-peker-diagram om du foretrekker).

```
(set-car! z2 (cdr z2))
```

(c) Definer en rekursiv prosedyre `nested-count` som tar et symbol og en potensielt nøstet liste som argument og returnerer antall forekomster av symbolet i listen. Kalleksempel:

```
(nested-match 'b '((b) ((b a) b) a))
→ 3
```

(d) Hva slags type prosess vil implementasjonen din av `nested-match` generere for kalleksemplet over?

## 2 let og lambda (7 poeng)

Skriv om følgende uttrykk til en ekvivalent form som bruker `lambda` i stedet for `let`. Oppgi også returverdien eller effekten uttrykkene har ved evaluering.

(a)
```
(let ((foo (list 1 2))
      (bar (* 2 2)))
  (cons bar foo))
```

(b)
```
(let ((foo (list 1 2)))
  (display foo)
  (newline)
  (let ((foo (cons 0 (cdr foo))))
    (display foo)))
```

# 3 Prosedyrer (12 poeng)

(a) Skriv en prosedyre `compose` som tar to prosedyrer som argument – la oss kalle dem `p1` og `p2` – og returnerer en ny prosedyre som anvender `p1` på resultatet av å anvende `p2` på argumentet sitt. Både `p1` og `p2`, og den nye prosedyren som returneres, forventer ett argument. Kalleksempel:

```
(define (add1 x) (+ x 1))

(define (add100 x) (+ x 100))

((compose add1 add100) 5) → 106
```

(b) Basert på `compose` skal du nå skrive en prosedyre `repeat` som tar en prosedyre `p` og et positivt heltall `n` som argumenter, og returnerer en ny prosedyre som anvender `p n` antall ganger. Kalleksempel:

```
((repeat add1 10) 20) → 30
```

(c) Definer `eval-infix` som skal ta som argument en tre-elements liste på formen *(arg1 operator arg2)*, og returnere verdien av å anvende operatoren i midten på de to operandene `arg1` og `arg2`. Kalleksempler:

```
(define exp1 (list 1 + 3))

(define exp2 (list 10 / 5))

(eval-infix exp1) → 4

(eval-infix exp2) → 2
```

# 4 Paradigmer og idiomer (25 poeng)

I denne oppgaven skal du skrive noen forskjellige versjoner av en enkel prosedyre `scale`. Argumentene skal være et tall `x` og en liste av tall `seq`, og returverdien en liste der hvert av tallene i `seq` har blitt multiplisert med `x`. Kalleksempel:

```
(define foo (list 1 2 3 4))

(scale 3 foo) → (3 6 9 12)
```

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

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

(c) Skriv en rent funksjonell versjon basert på høyereordens sekvensoperasjoner (det er greit å benytte seg av innebygde prosedyrer her).

(d) Skriv en destruktiv versjon `scale!` som modifiserer listeargumentet sitt.

(e) Skriv en strømversjon av løsningen din for deloppgave (b) over: Den skal ta en strøm av tall som argument og returnere en ny strøm av de skalerte elementene. Du kan her anta at hele grensesnittet for strømmer slik vi har brukt det i kurset er tilgjengelig.

(f) Med i utgangspunkt i kalleksempelet på sekvensen `foo` over, hvor mange `cons`-operasjoner vil utløses av å kalle de respektive `scale`-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.

## 5 Funksjonelle prosedyrer (6 poeng)

Innebygd i Scheme finnes to prosedyrer `for-each` og `map` som vi kan tenke at er implementert som vist under (litt forenklet). Bruk en setning eller to på å forklare hva som er likheten og, enda viktigere, forskjellen, på de to høyereordens prosedyrene slik de er vist her. En viktig forskjell mellom funksjonelle og ikke-funksjonelle prosedyrer gjenspeiles i forskjellen mellom `for-each` og `map`; forklar kort hva vi sikter til her.

```
(define (for-each proc items)
  (if (null? items)
      'done
      (begin (proc (car items))
             (for-each proc (cdr items)))))


(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))
```

## 6 Omgivelser (10 poeng)

Her skal vi jobbe med omgivelsesmodellen for evaluering. Tegn et omgivelsesdiagram som viser alle relevante rammer og bindinger etter at alle uttrykkene i følgende sekvens har blitt evaluert.

```
(define items '(a b))

(define (keeper x)
  (set! items (cons x items))
  items)

(define (make-keeper items)
  (lambda (x)
    (set! items (cons x items))
    items))

(define k1 (make-keeper '(c d)))

(k1 'e)

(keeper 'f)
```

# 7 Innkapsling (17 poeng)

Skriv en prosedyre `make-accumulator` som returnerer en ny prosedyre som innkapsler en lokal variabel `sum` (initialisert til 0) og lar oss legge til et tall (med beskjeden `'add`) eller trekke fra et tall (med beskjeden `'sub`). Prosedyren skal også ta beskjeden `'undo` som lar oss angre et gitt antall utførte kall og gjenopprette summen til hva den var før det. I alle tilfeller returneres den oppdaterte summen. Du kan gjerne definere ekstra hjelpeprosedyrer hvis du vil. Kalleksempler:

```
(define acc (make-accumulator))

(define acc2 (make-accumulator))

(acc 'add 5) → 5

(acc 'add 15) →  20

(acc 'add 80) →  100

(acc 'add 10) →  110

(acc 'sub 20) →  90

(acc 'undo 3) →  20

(acc2 'sub 5) →  -5
```

# 8 Evalueringsstrategier (7 poeng)

På slutten av kurset så vi på hvordan den metasirkulære Scheme-evaluatoren kunne endres til å implementere en form for såkalt *normal-order evaluation* (*lazy evaluation*) som standard evalueringsstrategi (for ikke-primitive prosedyrer). I den forbindelse valgte vi også å *memoisere* evalueringen av uttrykk som angir en prosedyres argumenter. Forklar kort hva som var motivasjonen for dette. Forklar også hvorfor det ikke var relevant å gjøre dette så lenge evaluatoren holdt seg til såkalt *applicative-order evaluation* (*eager evaluation*) som standard evalueringsstrategi.