Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2013-03-12
Sal (2) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	TER2 (TER3)
Tid	14-16
Kurskod	TDDC74
Provkod	TEN1
Kursnamn/benämning **Provnamn/benämning**	Programmering - abstraktion och modellering Skriftlig tentamen/duggor
Institution	IDA
Antal uppgifter som ingår i tentamen	3
Jour/Kursansvarig Ange vem som besöker salen	Jalal Maleki
Telefon under skrivtiden	Telefon: 013-28 19 63 Mobiltelefon: 070-607 19 63
Besöker salen ca kl.	ja
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, ankn. 2362, anna.grabska.eklund@liu.se
Tillåtna hjälpmedel	inga
Övrigt	
Vilken typ av papper ska användas, rutigt eller linjerat	Valfritt
Antal exemplar i påsen	

TDDC74 Programmering: Abstraktion och modellering
Dugga 3 (provkod TEN1), Tid: kl 08-10, Datum: 2013-03-12

Läs alla frågorna först och bestäm dig för den ordning som passar dig bäst.

Även om det i uppgiften står att du skall skriva en procedur/funktion, så får du gärna skriva ytterligare hjälpfunktioner som kan vara nödvändiga.

Det finns tre uppgifter i denna dugga. Poängen per uppgift/deluppgift anges i samband med varje uppgift.

Skriv tydligt för att öka läsbarheten. Använd **väl valda namn** på parametrar och **indentera** din kod. Sammanlagt två poäng är reserverade för väl-skriven kod, dvs, kod som innehåller bra namngivning, korrekt indentering, användning av dataabstraktion vid behov, och andra principer för bra Scheme-kod.

Betygsgradering: Det finns tre duggor i kursen. Varje dugga ger 12p, dvs totalt 36p för alla tre. För att passera en dugga krävs minst 3p på duggan. Totalt skall du på de tre duggorna för betyget 3 ha minst 18p, för betyget 4 minst 23p och för betyget 5 minst 27p.

Lycka till!

Uppgift 1

(2 poäng) Studera följande kod som implementerar enkla objekt (lampor). En lampa kan antingen vara på eller av.

```
(define *lamps* ())

(define (make-lamp)
   (define status 'off)

   (define (turn-me-off)
      (set! status 'off))

   (define (turn-me-on)
      (set! status 'on))

   (define (dispatch message)
      (cond ((eq? message 'status) status)
            ((eq? message 'on)
             (turn-me-on))
            ((eq? message 'off)
             (turn-me-off))
            (else (error "Invalid message: " message))))

   (set! *lamps* (cons dispatch *lamps*))

   dispatch)
```

Som du har observerat har man en global variabel ***lamps*** där alla lamp-objekt sparas i en lista. På så sätt kan man alltid få tag i alla lampor som skapats.

Antag att vi har skapat ett antal lampor genom att anropa **make-lamp** några gånger och sedan ändrat status på vissa till **on**. Din uppgift är nu att skriva en procedur som går genom ***lamps*** och ändrar varje lamps status, dvs om status är **off** då ändras det till **on** annars ändras det från **off** till **on**.

Uppgift 2

Rita omgivningsdiagram som fås efter de följande uttrycken är evaluerade i den ordning som de förekommer nedan. När värdet på en variabel ändras kryssa över det gamla värdet och skriv det nya bredvid.

```
(define age 1)
```

```
(lambda (age) (+ age age))
```

```
(set! age (+ age 1))
```

```
((lambda (age) (set! age (+ age 1))) age)
```

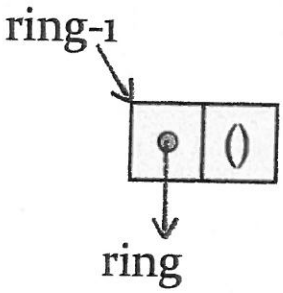
```
(define f
  (let ((state 0))
    (lambda (n)
      (if (= state 0)
          n
          (begin
            (set! state (- 1 state))
            (+ state (f (- n 1))))))))
```

```
(* (f 1) (f age))
```

Uppgift 3

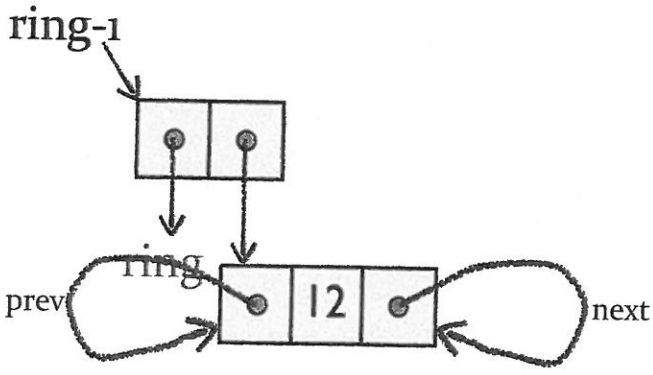
I den här uppgiften skall du jobba med en så kallad ringstruktur. Ringen kommer att innehålla noll eller flera element som pekar fram och tillbaka till varandra. På så sätt är det möjligt att flytta sig både framåt och bakåt i strukturen. Varje element i ringen kommer att innehålla ett värde (ett heltal) och två pekare: en som pekar till nästa element och en som pekar till föregående element i ringen. Du måste se till att elementen hålls i stigande ordning (när man flyttar sig framåt i strukturen). Det första elementet kommer att innehålla det minsta värdet och det sista elementet i ringen det största värdet i ringen.

En ring (ring-1) som inte har några element ser ut så här,



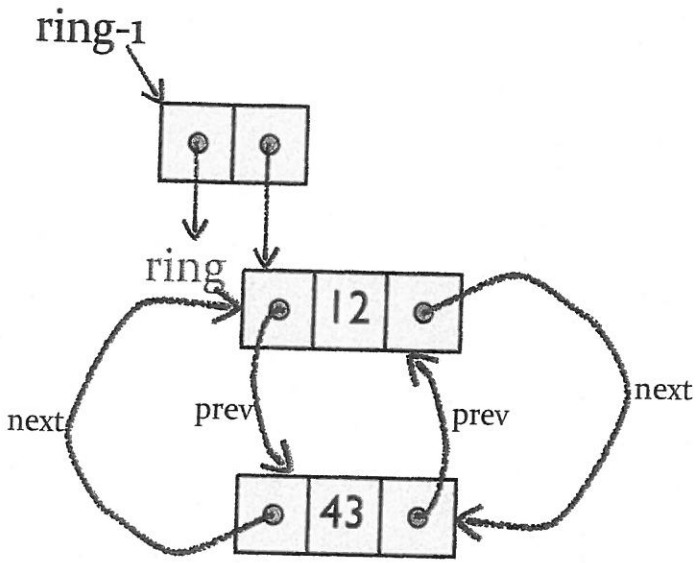
Dvs, en **mcons**-struktur där **mcar**-delen innehåller symbolen **ring** och **mcdr**-delen pekar på första elementet i ringen. För närvarande finns inga element varför **mcdr**-delen är lika med **()**.

Vi lägger in ett element i ringen genom att anropa **(ring-insert! ring-1 12)** som skapar ett element med värdet 12:



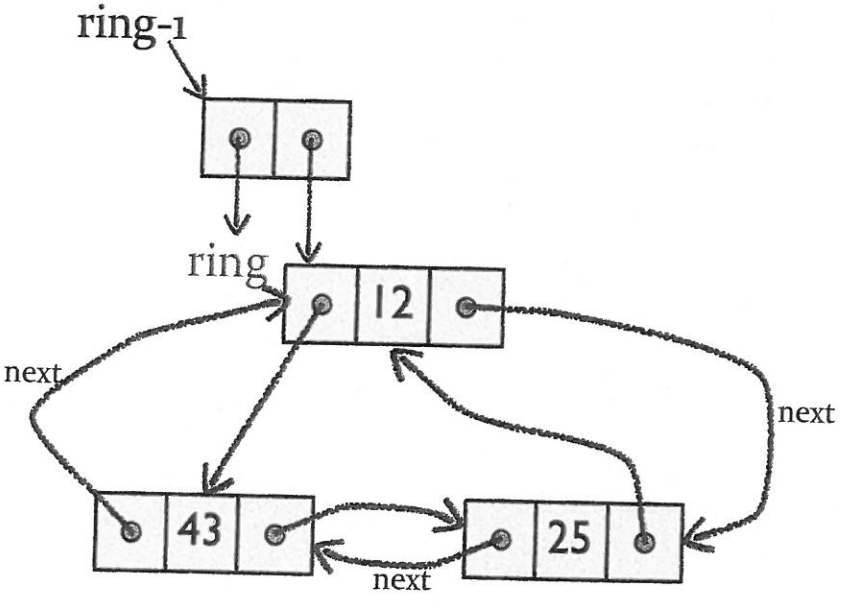
Observera hur pekarna (förkortad **next** och **prev**) pekar ut nästa resp föregående element. Än så länge pekar de på ett och samma element (elementet själv).

Efter ytterligare anrop **(ring-insert! ring-1 43)** får vi följande struktur.



Här ser vi att elementet med värdet 12 pekar ut elementet med värdet 43 som sitt nästa element och elementet med värdet 43 pekar ut första elementet som sitt nästa element.

Sedan anropar vi **ring-insert!** igen med värdet 25, **(ring-insert! ring-1 25)**. Observera att elementet med värdet 25 har hamnat mellan 12 och 45 eftersom vi ville hålla ringen i ökande ordning.



Som Ni ser från exemplen ovan, har en ring ett antal element där sista elementet pekar tillbaka till det första och på så sätt bildas en cirkulär struktur. Vi definierar ring som en dataabstraktion. Här är några av de procedurer som ingår i implementationen av denna dataabstraktion:

```
(define (make-ring)
   (mcons 'ring ()))


(define (ring-label ring)
   (mcar ring))


(define (ring-first ring)
   (mcdr ring))


(define (empty-ring? ring)
   (null? (mcdr ring)))
```

Komplettera och använd dessa vid behov i samband med din lösning till uppgift (b) nedan.

a) (2 poäng) Skapa en dataabstraktion för elementen i ringen. Ett sådant element har tre delar som kan lämpligen kallas för **previous**, **value** och **next**. Dataabstraktionen skall inkludera lämpliga konstruktor-, selektor-, igenkännar- och mutatorfunktioner för den tredelade strukturen.

b) (3 poäng) Skriv proceduren (**ring-insert!** *ring value*) som lägger in ett element som innehåller värdet *value* på rätt ställe i ringen så att den ökande ordningen behålls. De värden som finns i ringens element är inte unika, dvs, det kan finnas fler än ett element med ett och samma värde.