Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2014-03-06
Sal (2) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	TER2 TERC
Tid	8-10
Kurskod	TDDC74
Provkod	KTR2
Kursnamn/benämning **Provnamn/benämning**	Programmering - abstraktion och modellering Frivillig dugga
Institution	IDA
Antal uppgifter som ingår i tentamen	4
Jour/Kursansvarig Ange vem som besöker salen	Johannes Schmidt
Telefon under skrivtiden	0725-721800
Besöker salen ca kl.	ja (ca 08:30)
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 2, kl 8—10, 6 mars 2014

Läs alla frågorna först, och bestäm dig för i vilken ordning du vill lösa uppgifterna.

Skriv tydligt och läsligt. Använd **väl valda namn** på parametrar och **indentera** din kod.

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

OBS! Du får använda `define` för att definiera procedurer och namn. Men du får inte använda det (eller `set!`) för att ändra på/uppdatera variabelvärden. Tanken är att lösningarna ska vara strikt funktionella.

Betyg: Det finns tre duggor i kursen. På varje dugga kan man få som mest 12p (så totalt 36p för alla tre).

För att bli godkänd på en dugga krävs minst 3p.

För att kunna få betyget 3 på duggorna måste du sammanlagt (på alla tre duggor) ha fått minst 18p, för betyget 4 gäller minst 23p och för betyget 5 minst 27p.

För att bli godkänd på duggaserien måste du dessutom ha godkänt på samtliga duggor.

Lycka till!

Uppgift 1 (3 poäng)

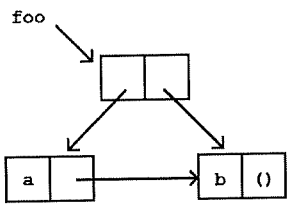
a) Antag att vi har evaluerat följande uttryck i Scheme.

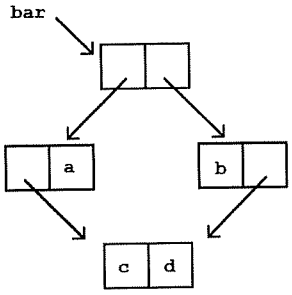
```
(define a 57)
(define b (lambda (x) 1))
(define x (cons a b))
(define y (cons 'a 'b))
(define z (list 1 2 3))
```

Rita box-pointer-diagram för dessa strukturer. Var noga med pekarna!

```
x
y
z
(list x y z)
(cons a 'a)
(cons (b y) (cons 5 z))
```

b) Skriv Schemekod som skapar dessa två strukturer:





Uppgift 2 (3 poäng)

En mängd är en oordnad samling av unika element. Följande är exempel på mängder:
$\{\}$ är tomma mängden
$\{1, 2, 3\}$ är en mängd av tre heltal
{äpple, apelsin, persika, melon} är en mängd som innehåller namnen på några frukter.

Mängder kan i Scheme representeras som listor, t.ex,

```
(define tomma-mangden ())
(define tre-tal '(1 2 3))
(define fyra-tal '(2 3 4 5))
(define frukt '(apple apelsin persika melon))
(define daggdjur '(hast elefant val manniska))
(define havsdjur '(delfin val haj))
```

Det finns ett par viktiga skillnader mellan listor och mängder: ordning och dubletter. I en mängd spelar det ingen roll vilken ordning elementen kommer i. Det är också så att elementen i en mängd alltid är unika (så inga dubletter får förekomma i vår represenation).

Uppgift Vi vill beräkna *unionen* av två mängder. Unionen av mängderna A och B är den mängd som består av alla element i A och alla element i B.

Vi skulle kunna använda `append` för att hitta något som liknar unionen av mängderna (alla element). Men isåfall skulle vi riskera att få med dubletter.

Skriv en Scheme-procedur `union` som beräknar unionen av två mängder (representerade som listor). Inga dubletter får förekomma i slutresultatet.

```
> (union '(1 2 3) '(2 3 4))
(1 2 3 4)
> (union '(1 2 3) '(4 3 2))
(1 2 3 4)
> (union '() (1))
(1)
```

OBS! Det spelar ingen roll vilken ordning elementen kommer i. Din lösning kan mycket väl ha elementen i en annan ordning än den i exemplet.

Ni kan anta att indata till `union` är korrekta mängder. Ingen av listorna som vi skickar in till funktionen kan alltså innehålla dubletter.

Uppgift 3 (3 poäng)

Vid föreläsning 5 visade vi de generella `accumulate`, `map`, och `enumerate`-mönstren:

```
(define (accumulate proc null-value L)
  (if (null? L)
      null-value
      (proc (car L)
            (accumulate proc null-value (cdr L)))))

(define (map proc L)
  (if (null? L)
      ()
      (cons (proc (car L)) (map proc (cdr L)))))

(define (enumerate low high)
  (if (> low high)
      ()
      (cons low (enumerate (+ low 1) high))))
```

Antag dessutom att du har en funktion `fib` som tar ett tal `n` och ger det n:te Fibonacci-talet.

Uppgift Skriv uttryck (som använder funktionerna ovan) som

1. genererar en lista med alla Fibonaccital mellan 5 och 12 (inklusive gränserna för intervalet),
2. beräknar summan av alla Fibonaccital mellan 5 och 12 (inklusive gränserna),
3. beräknar summan av alla udda Fibonaccital mellan 5 och 12 (inklusive gränserna).

Uppgift 4 (3 poäng)

Hittills har vi stött på vanliga heltal i Scheme. Vi introducerar nu rationella tal (bråkform) och komplexa tal, och ger dem en egen representation.

Ett rationellt tal representeras av ett par heltal (a, b) (täljare och nämnare). Ett komplext tal representeras också av ett par heltal (a, b) (det vi vanligen skriver $a + bi$). Ett ord för beloppet av ett tal är *magnitude*. Detta är

- $mag(x) = x$ om $x > 0, -x$ annars, för reella tal x.
- $mag((a, b)) = mag(a/b)$ för rationella tal (a, b).
- $mag((a, b)) = \sqrt{(a^2 + b^2)}$ för komplexa tal (a, b).

Vi vill nu skapa en procedur som tar ett tal - av något av slagen ovan - och beräknar beloppet av det.

För att se vilken typ av tal vi får in, kopplar vi varje uppsättning siffror till en *etikett* (*tag*) som talar om vad för slags data det är. Etiketterna är `number`, `rational` och `complex`.

Här är de tillhörande konstruktorerna:

```
(define (make-number n)        (cons 'number   n))
(define (make-rational a b)    (cons 'rational (cons a b)))
(define (make-complex a b)     (cons 'complex  (cons a b)))
```

Uppgift Skapa predikaten `number?`, `rational?` och `complex?` som kan skilja på de tre sorternas tal. De ska fungera enligt följande exempel:

```
(define x (make-number -4))
(define y (make-rational 3 -5))
(define z (make-complex 3.4 -5.55))

> (number? x)
#t
> (rational? x)
#f
> (complex? y)
#f
> (complex? z)
#t
```

Uppgift Vi implementerar *mag* som den ovan. För det ändamålet behöver vi kunna ta ut n ur ett av våra "number" samt a- och b-delarna från komplexa respektive rationella tal. Implementera `get-a`, `get-b`, `get-n` som gör att följande kod fungerar på önskat vis:

```
(define (mag number)
  (cond
    ((number? number)    (abs (get-n number)))
    ((rational? number) (abs (/ (get-a number)
                                (get-b number))))
    ((complex? number)  (sqrt (+ (square (get-a number))
                                 (square (get-b number)))))
    (else 'unkown-number)))
```