Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2012-01-10
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	T1
Tid	14-18
Kurskod	TDDC74
Provkod	TEN1
Kursnamn/benämning **Provnamn/benämning**	Programmering - abstraktion och modellering Skriftlig tentamen/duggor
Institution	IDA
Antal uppgifter som ingår i tentamen	6
Jour/Kursansvarig Ange vem som besöker salen	Anders Haraldsson
Telefon under skrivtiden	ankn. 1403 eller 0705-147709
Besöker salen ca kl.	ca. kl. 15
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

AID-nummer:	Datum: 2012-01-10	1
Kurskod: TDDC74	Provkod:TEN1	

Tekniska högskolan vid Linköpings universitet
Institutionen för datavetenskap
Anders Haraldsson

TDDC74 Programmering, abstraktion och modellering

Tentamen

Tisdag 10 januari 2012 kl148-18

Uppgifterna löses direkt på denna uppgiftslapp, som lämnas in i sedvanligt tentamensomslag. Räcker inte utrymmet kan du komplettera med lösa papper.

Skriv tydligt så att inte dina lösningar missförstås. Använd väl valda namn på parametrar etc.

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

Enligt den nya standarden i Scheme/Racket använder man en ny uppsättning funktioner för cons-celler man vill ändra pekare i (dvs muterbara cons-celler), dvs mcons, mcar, mcdr, set-mcar!, set-mcdr! etc. Det går lika bra att använda cons, car, cdr, set-car!, set-cdr!.

Betygsgradering:

12-16,5	betyg 3
17-20,5	betyg 4
21-24	betyg 5

Lycka till

Uppgift 1. Beräkning av uttryck och quote (2 poäng)

Vi ger följande uttryck för beräkning. Vilket värde skrivs ut eller om det blir fel, beskriv typen av fel.

```
> (define x 1)

> (define y z)

> (define z 2)

> (define v (list 'y x z))

> v                                  =>

> (x y z)                            =>

> (quote (list x y z))               =>

> (define (f x y) (+ x y z))

> (f x z)                            =>

> (define (g x a)
    (define (f y) (+ y a))
    (f (+ x z))))

> (g 3 x)                            =>

> (f 1)                              =>

> (define k (lambda (x y) (y (y x))))

> (k 20 (lambda (x) (+ x 10)))       =>

> (k f list)                         =>
```

0-1 rätt	0p
2-3 rätt	0,5p
4-5 rätt	1,0p
6-7 rätt	1,5p
8 rätt	2,0p

Uppgift 2. Rekursiva procedurer för tal (4 poäng)

McLaurin-serier. Vissa funktioner kan beräknas genom att summera en oändlig serie. Vi skall här skriva en funktion, som summerar en ändlig serie, där vi anger hur många termer som skall summeras.

Funktionen *sin* kan beräknas med

$$\sin x = \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

3b. (1p) Skriv först en fakultetsfunktion (fak n), som beräknar *n!*.

```
(define (fak n)
```

Beskriver din funktion en *rekursiv* eller *iterativ* (linear) *processkösning*? Motivera.

3c. (3p) Skriv sedan en funktion (sin n x), som beräknar *sin x* med hjälp av McLaurin-serien med *n* termer[1].

```
(define (sin n x)
```

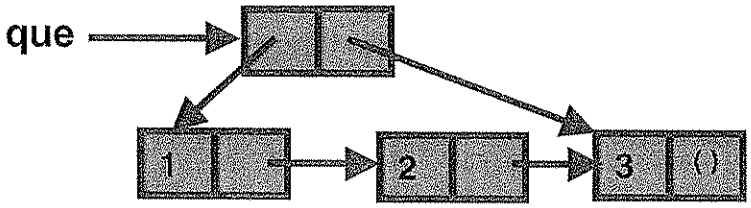
[1] Scheme har funktionen (expt x n) som beräknar x^n.

Uppgif 3. Listor, cons-celler och pekare (3 poäng)

3a. (1p)Vad blir värdet i parentesformat av följande uttryck. Rita även upp värdet med den grafiska representationen med cons-celler och pekare (box and pointer notation).

```
(define test (cons (cons 'x '()) (list 'a 'b)))
test =>
```

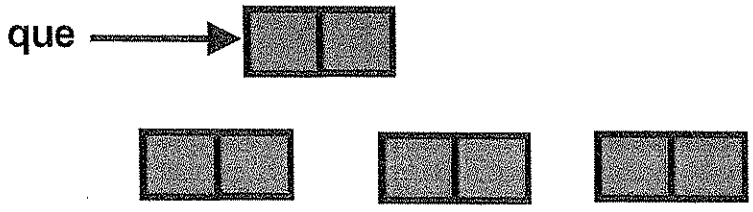
3b. (1p) En kö representeras med ett köhuvud och en lista med köelement. Köhuvudet anger första resp. sista köelementet. Skriv ett Scheme-uttryck som skapar följande köstruktur:



3c. (1p) Vad händer efter det att följande två Scheme-uttryck har beräknats.

```
(set-car! (cdr  que) 4)
(set-cdr! (car que) (cdr (cdr (car que))))
```

Rita in pekare och värden hur strukturen nu ser ut:



Blank sida. Kan användas om inte utrymmet räcker till. Ange noggrant vilka uppgifter som finns på denna sida.

Uppgift 4. Rekursiva procedurer för listor (7 poäng)

4a. (2p) Skriv en *rekursiv* procedur (lägg-in *tal sorterad-lista*), som tar ett tal och en sorterad lista (i stigande ordning, dvs minsta talet först) med tal som placerar in ett nytt element. Elementen placeras in på sorterad plats.

```
> (lägg-in 2 '(1 3 4))        => (1 2 3 4)
> (lägg-in 0 '(1 3 4))        => (0 1 3 4)
> (lägg-in 100 '(1 3 4))      => (1 3 4 100)
```

Beskriver din lösning en *rekursiv* eller *iterativ (linear) processlösning*? Motivera.

```
(define  (lägg-in e sort-l)
```

4b. (3p) Skriv en funktion (max-djup l), som i en godtycklig lista, där elementen i sin tur kan vara listor (men ej punkterade par), och som returnerar det maximala djupet i listan. Med det menar vi maximala antalet listor i listor tills vi kommer ned till en icke-lista.

```
> (max-djup '())                => 1
> (max-djup '(a b c))           => 1
> (max-djup '((a b) c))         => 2
> (max-djup '(a (b (c d) e) (f g)))  => 3
```

```
(define (max-djup l)
```

Uppgift 4. Rekursiva procedurer för listor, forts

4c. (2p) Fortsättning på uppgift 4a. Skriv en procedur (lägg-in-order *element list ordnings--fn*), som dessutom tar en ordningsfunktion, som anger hur sorteringen skall gå till (stigande eller fallande eller på annat sätt). Ordningsfunktionen skall ta två godtyckliga element som argument och returnera ett sant värde om det första argumentet skall komma före det andra argumentet i i sorteringsordning. Listan kan nu bestå av element av godtycklig typ, ej bara som tal i uppgift 3a. Elementen i den sorterade listan uppfyller alla den sorteringsordning, som ordningssfunktionen anger.

```
> (lägg-in-order  2 '(1 3 4) <)          => (1 2 3 4)
> (lägg-in-order  5 '(10 7 4 2) >)       => (10 7 5 4 2)
```

```
(define (lägg-in-order e sorted-l order-fn)
```

Med hjälp av lägg-in-fn komplettera (vad skall ? ersättas med) definitionerna av följande procedurer:

Definiera en procedur lägg-in-absolut, som i en godtycklig lista med tal lägger in tal sorterade efter absolutbeloppet.

```
> (define (lägg-in-absolut tal lista-med-tal)
      (lägg-in-fn tal lista-med-tal ?))

> (lägg-in-absolut 3 '(1 -2 -4 5))       => (1 -2 3 -4 5)
> (lägg-in-absolut -3 '(1 -2 -4 5))      => (1 -2 -3 -4 5)

? =
```

Definiera en funktion lägg-in-dellista, som i en lista med dellistor, som är sorterade på listornas längd[2]med den längsta listan först, lägger i en ny dellista.

```
> (define (lägg-in-dellista lista lista-med-dellistor)
      (lägg-in-fn lista lista-med-dellistor ?))

> (lägg-in-dellista '(1 2) '((1 2 3 4 5) (1 2 3 4) (1)))  => ((1 2 3 4 5) (1 2 3 4) (1 2) (1))

? =
```

[2] Funktionen length räknar ut längden av en lista.

Uppgift 5. Abstraktion och objektorientering (4 poäng)

Givet en databas med information om studenter, kurser och resultat. För varje student finns namn och personnummer, för varje kurs finns kod och kursnamn samt för varje resultat finns personnummer, kurs och betyg. Vi har introducerat 3 abstrakta datatyper för **student**, **kurs** och **resultat**. Allt lagras i en databas, i form av en associationslista. Nedan finns kod för hantering av databasen och primitiver för de abstrakta datatyperna. I representationen för de abstrakta datatyperna har man använd metoden med typmärkning.

Primitiver för hantering av märkning:

```
(define märk cons)
(define typ car)
(define ta-bort-märke cdr)
```

Primitiver för att skapa och lägga in data i en databas

```
(define (skapa-db)
  (let ((db '()))
    (define (dispatch msg index . args)
            (cond ((eq? msg 'lägg-in)
                    (set! db (cons (cons index (car args)) db)))
                  ((eq? msg 'ta-fram)        Se uppgift 5b. Kompletterah är med kod
                  (else (error "fel meddelande"))))))

(define (lägg-i-databas db index . info) (db 'lägg-in index info))
```

Med dessa procedurer kan vi nu skapa en databas:

```
(define Y-programmet (skapa-db))
(define MED-programmet (skapa-db))
```

och sedan lägga in data i en databas givet ett index:

```
(lägg-i-databas Y-programmet 'studenter
        (skapa-student '(Anders Andersson) '900202-0202)
        (skapa-student '(Lisa Andersson) '910303-0303)
        (skapa-student '(Anders Eriksson) '920404-0404))

(lägg-i-databas Y-programmet 'kurser
        (skapa-kurs 'TDDX00 'Pram)
        (skapa-kurs 'TDDX01 'Analys))

(lägg-i-databas Y-programmet 'resultat
        (skapa-resultat '900202-0202 'TDDX00 'betyg3)
        (skapa-resultat '910303-0303 'TDDX00) 'betyg4)
        (skapa-resultat '900202-0202 'TDDX02) 'betyg5))
```

Uppgift 5. Abstraktion och objektorientering, forts

Vi definierar även följande konstruktorer för de abstrakta datatyperna:

```
(define (skapa-student namn pnr) (märk 'student (list namn pnr))
(define skapa-kurs (kod kurs-namn) (märk 'kurs (list kod kurs-namn)))
(define skapa-resultat (pnr kod betyg) (märk 'resultat (list pnr kod betyg)))
```

5a. (1p) Vi behöver selektorer för att ta fram personnummer, namn, kurskod, kursnamn och betyg . Definiera de två första. De övriga behöver ej definieras.

För att ta fram ett personnumer

```
(define (pnr obj)
  ; personnummer finns för student och resultat, obj kan vara antigen en student eller ett resultat.
```

Komplettera här med kod

För att ta fram ett namn.

```
(define (namn obj)        ; namn finns för student
```

Komplettera här med kod

```
(define (kod obj)         ; kod finns för kurs och resultat
(define (kurs-namn obj)   ; kursnamn finns för kurs
(define (betyg obj)       ; betyg finns för resultat
```

Vi skal t.t ex. kunna göra följande:

```
(define per (skapa-student '(Per Persson) '901212-0202))
(define res (skapa-resultat '910101-0202 'TDDX00 'betyg3))
(pnr per)     => 901212-0202
(pnr res)     => 910101-0202
(namn per)    => (Per Persson)
```

5b.(1p) Komplettera med kod i proceduren skapa-db för att givet ett index ta fram den lagrade informationen. Vi vill t.ex. kunna skriva följande uttryck för att få fram Y-programmets kurser:

```
(Y-programmet 'ta-fram-info 'kurser) => ((kurs TDDX00 Pram) (kurs TDDX01 Analys))
```

5c (2p) Definiera en procedur (plocka-ut index db fn), som givet index och en databas finner den lagrade informationen och ur den plockar ut alla poster som uppfyller funktionen *fn*. I nedanstående exempel vill vi plocka ut alla poster i student-listan för studenter som heter Anders i förnamn.

```
(plocka-ut 'studenter Y-programmet (lambda (e) (eq? (car (namn e)) 'Anders)))
=> ((student (Anders Andersson) 800202-0202) (student (Anders Eriksson) 800404-0404))

(define (plocka-ut index db fn)
```

Uppgift 6. Procedurobjekt, omgivningsdiagram (4 poäng)

6a. (2p) I samband med omgivningsdiagram förklara

- bindning (*binding*)
- ram (*frame*)
- omgivning (*environment*)
- procedurobjekt (*procedure object*)

Omgivningsdiagrammet används då man i Scheme ger följande uttryck:

- definition: (define variabel uttryck) i en omgivning En
- tilldelning: (set! variabel uttryck) i en omgivning En
- lambda-uttryck: (lambda formella-parametrar kropp) i en omgivning En
- funktionsapplicering: (procedur-uttryck aktuella-parametrar ...) i en omgivning En

Nedanstående exempel använder ovanstående begrepp. Förklara resp. begrepp då motsvarande begrepp utförs i exemplet och beskriv vad som händer med omgivningsdiagrammet. Rita upp omgivningsdiagrammet, visa ändringarna och ange i vilken ordning ramarna skapas.

```
> (define a 20)
> (define f (lambda (x y) (set! x (+ x 1)) (+ x y)))
> (f a 5) => ?
```

Uppgift 6. Procedurobjekt och omgivningsdiagram, forts

6b. (2p) Följande Scheme-uttryck beräknas. Rita hur omgivningsdiagrammet ser ut efter det att alla uttrycken har beräknats. Vilka värden skrivs ut och rita omgivningsdiagrammet?

```
(define  counter 0)

(define memory
        (let ((mem'()))
          (lambda (n) (set! counter (+ counter 1))
                      (set! mem (cons n mem))
                      mem)))

(memory 20) => ?
(memory 10) => ?
a => ?
```