Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2013-06-07
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	KÅRA
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	Jalal Maleki
Telefon under skrivtiden	0706-071963
Besöker salen ca kl.	ja
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, ankn. 2362, anna.grabska.eklund@liu.se
Tillåtna hjälpmedel	
Övrigt	
Vilken typ av papper ska användas, rutigt eller linjerat	
Antal exemplar i påsen	

Tentamen

TDDC74 Programmering: Abstraktion och modellering
Provkod TEN1, Tid: kl 14-18, 2013-06-07, Kåra

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.

Skriv tydligt för att öka läsbarheten. Använd **väl valda namn** på parametrar och **indentera** din kod. Sammanlagt 2 poäng tilldelas väl-skriven kod, dvs, kod som innehåller bra namngivning, korrekt indentering, användning av dataabstraktion vid behov, och andra principer för bra Scheme-kod.

Det finns sex uppgifter i tentan. Poängen per uppgift/deluppgift anges i samband med varje uppgift.

Betygsgradering: För betyget 3 krävs minst 12p, för betyget 4 minst 16p och för betyget 5 minst 19p. Högst möjliga antal poäng är 24.

Lycka till!

Uppgift 1 - föränderliga liststrukturer

(4 poäng) Antag att x och y är definierade enligt nedan:

```
(define x 3)
(define y 6)
```

Rita följande strukturerna som mcons-lådor och pekare:

1. `(define k (mcons x (mcons x y)))`
2. `(define m (mcons x y))`
3. `(define n (mcons (mcar k) (mcar (mcdr k))))`
4. `(define p (mcons m n))`
5. `(define q (mcons m n))`
6. `(set-mcar! (mcar q) 8)`
7. `(set-mcdr! q 12)`
8. `(set-mcdr! p (mcar (mcar q)))`

Uppgift 2 - enkel iteration/rekursion

(4 poäng) Skriv en fakultetsfunktion `(fak n)`, som beräknar $n!$ ($n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$).
T ex *0! = 1 per definition, 4! = 4·3·2·1=24, 5! = 5·4·3·2·1 = 120*
Redogör sedan för huruvida din funktion är en rekursiv eller iterativ processlösning? Motivera.

Uppgift 3 - iteration/rekursion

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

Funktionen *sin* kan beräknas med hjälp av följande approximation:

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

(4 poäng) Skriv sedan en funktion `(sin n x)` som beräknar $\sin x$ med hjälp av McLaurin-serien med n termer. Du får gärna använda Schemefunkionen `expt - (expt m n)` - som beräknar m^n och naturligtvis fakultetsfunktionen från tidigare uppgiften.

Uppgift 4 - listhantering

(4 poäng) Skriv en Schemefunktion **merge** med två parametrar där båda antas vara ordnade listor av heltal. **merge** skapar en ny ordnad lista av heltal som innehåller alla element som ingår i de två parametrarna. Följande exempel förklarar **merge** ytterligare.

```
> (merge '() '())
()

> (merge '() '(1 2 3))
(1 2 3)

> (merge '(1 2 6) '(2 3 4 5))
(1 2 2 3 4 5 6)
```

(2 poäng) Använd substitutionsmodellen för att redovisa hur din implementation av merge beräknar `(merge '(4 7) '(1 3 8))`.

Uppgift 5 - dataabstraktion och listhantering

(4 poäng) En datamängd över flygresor har strukturerats enligt följande:

```
(define flights
  (list
   ;;       airline orig dest flt# dprt arrv
   ;;
   (make-flight 'AS 'YVR 'SEA 2039 0930 1020)
   (make-flight 'AS 'SEA 'YVR 2126 1220 1310)
   (make-flight 'AS 'SEA 'ANC   85 1150 1430)
   (make-flight 'AS 'ANC 'FAI   93 1530 1620)
   (make-flight 'NW 'SEA 'MSO 2199 1700 2025)
   (make-flight 'UA 'YVR 'SEA 1483 0605 0650)
   (make-flight 'UA 'ANC 'FAI 1744 2220 2314)
   (make-flight 'UA 'SEA 'ANC 1760 1410 1636)
   (make-flight 'UA 'SEA 'SFO 1203 0630 0835)
   (make-flight 'UA 'SEA 'SFO 1764 1130 1334)
   (make-flight 'UA 'SEA 'OAK 1143 1228 1444)
   (make-flight 'UA 'SEA 'ORD  154 0700 1248)
   (make-flight 'UA 'SFO 'SEA 1706 1030 1234)
   (make-flight 'UA 'SFO 'RDD 7005 1420 1535)
   (make-flight 'UA 'SFO 'EWR   12 0800 1622)
   (make-flight 'UA 'RDD 'SFO 7014 0645 0800)
   (make-flight 'UA 'EWR 'LHR  906 1855 0645)
   (make-flight 'UA 'ORD 'EWR  362 1330 1648)))
```

make-flight tar följande information och skapar en ny flygresa: förkortningen för flygbolaget som äger planet, förkortningen för den flygplats där resan börjar, förkortningen för den flygplats där resan slutar, flygresans nummer, avgångstid, ankomsttid. Konstruktorn definierar vi så här:

```
(define make-flight
  (lambda (carrier origin dest flight-no depart arrive)
    (list carrier origin dest flight-no depart arrive)))
```

T ex en resa som **my-flight** skulle kunna definieras som:

```
(define my-flight
        (make-flight 'UA 'EWR 'LHR  906 1855 0645))
```

Antag nu att följande selektorer är definierade för att hämta olika information för en given flygresa:

(flight-carrier my-flight) returnerar **UA**
(flight-origin my-flight) returnerar **EWR**
(flight-destination my-flight) returnerar **LHR**
(flight-no my-flight) returnerar **906**
(flight-depart my-flight) returnerar **1855**
(flight-arrive my-flight) returnerar **645**

Definiera proceduren **columns** med två parametrar: en lista såsom flygresor och en lista på godtyckligt antal selektorer för elementen i första listan. **columns** returnerar enbart de kolumner som anges av selektorlsistan. Följande exempel förklarar detta ytterligare.

```
> (columns flights (list flight-carrier flight-origin))
((AS YVR)
 (AS SEA)
 (AS SEA)
 (AS ANC)
 (NW SEA)
 (UA YVR)
 (UA ANC)
 (UA SEA)
 (UA SEA)
 (UA SEA)
 (UA SEA)
 (UA SEA)
 (UA SFO)
 (UA SFO)
 (UA SFO)
 (UA RDD)
 (UA EWR)
 (UA ORD))

> (columns flights (list flight-carrier flight-origin flight-no))
((AS YVR 2039)
 (AS SEA 2126)
 (AS SEA 85)
 (AS ANC 93)
 (NW SEA 2199)
 (UA YVR 1483)
 (UA ANC 1744)
 (UA SEA 1760)
 (UA SEA 1203)
 (UA SEA 1764)
 (UA SEA 1143)
 (UA SEA 154)
 (UA SFO 1706)
 (UA SFO 7005)
 (UA SFO 12)
 (UA RDD 7014)
 (UA EWR 906)
 (UA ORD 362))
```

Uppgift 6 - omgivningsdiagram och objekt

En stack (eller en hög) är en struktur som innehåller 0 eller flera element. Nya element läggs (PUSH) ovanför det senaste elementet, och man kan enbart ta bort (POP) det senast inlagda elementet från stacken. En hög (eller en stack) är tom från början. Så här kan vi föreställa oss att man kan definiera och använda en stack:

Implementera stack som objekt:

```
(define (make-stack)
  (let ((content ()))
    (define (push x)
      (set! content (cons x content)))
    (define (pop)
      (if (null? content)
          (error "Empty stack -- POP")
          (let ((top (car content)))
            (set! content (cdr content))
            top)))
    (define (initialize)
      (set! content '())
      'done)
    (define (dispatch m)
      (cond ((eq? m 'push) push)
            ((eq? m 'pop) (pop))
            ((eq? m 'initialize) (initialize))
            (else (error "Unknown request -- STACK" m))))
    dispatch))

(define (pop stack)
  (stack 'pop))

(define (push stack x)
  ((stack 'push) x))
```

Och här följer hur den är tänkt att användas.

```
> (define s (make-stack))
> (push s 2)
> (push s 4)
> (pop s)
4
```

(4 poäng) Rita de omgivningsdiagram som avbildar hur ovanstående uttryck evalueras.