Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2011-02-18
Sal (2) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	TER1 TER2
Tid	14-16
Kurskod	TDDC74
Provkod	KTR2
Kursnamn/benämning **Provnamn/benämning**	Programmering - abstraktion och modellering Frivillig dugga
Institution	IDA
Antal uppgifter som ingår i tentamen	6
Jour/Kursansvarig Ange vem som besöker salen	Anders Haraldsson
Telefon under skrivtiden	0705-147709
Besöker salen ca kl.	15.00
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	

AID-nummer:	Datum: 2011-02-18	1
Kurskod: TDDC74	Provkod: KTR2	

Tekniska högskolan vid Linköpings universitet
Institutionen för datavetenskap
Anders Haraldsson

TDDC74 Programmering, abstraktion och modellering

DUGGA 2

Fredag 18 feb 2011 14-16

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 ytterliggare hjälpfunktioner, som kan vara nödvändiga.

Betygsgradering: Det är tre duggor. Varje dugga ger 12p, dvs totalt 36p. För att passera en dugga krävs minst 3p på duggan. Totalt skall du på de tre duggorna för betyget 3 ha minst 20p. För betyget 4 minst 25p och för betyget 5 minst 30p.

Lycka till

Uppgift 1. (1 poäng) Quote

Guvet följande:

```
(define a 10)
(define b 20)
```

Vad blir värdet av följande uttryck

'(list (cons a b)) =>

(list (cons 'a 'b)) =>

(list (cons 'a b)) =>

(list '(cons a b)) =>

(list 'cons 'a 'b) =>

(list cons 'a 'b) =>

Poäng på uppgiften:

0-2 rätt	0p
3- 4 rätt	0,5p
5-6 rätt	1.0p

Uppgift 2. (1p) Rekursiv och iterativ processlösning

Vi har nedan givet två olika lösningar på samma problem. Vi har en funktion, som ökar varje element med 1. Vi har en lösning med *rekursiv processlösning* (linear reccursive model) och en med *iterativ processlösning* (iterative recurisve model) . En av dessa kommer att returnera ett resultat, där listan är omvänd. Använd substitutionsmetoden för motivera vilken av lösningarna, som ger en rättvänd resp. en omvänd lista som resultat. Använd argumentet listan (3 1 5) för resp. funktion.

```
(define (add-one-1 lst)
    (if (null? lst)
       '()
       (cons (+ 1 (car lst)) (add-one-1(cdr lst)))))

(define (add-one-2 lst)
  (define (help lst res)
    (if (null? lst)
        res
        (help (cdr lst) (cons (+ 1 (car lst)) res))))
  (help lst '()))
```

Är det den rekursiva eller iterative processlösningen som vänder på listan?

Motivera med substitutionsutveckling:

(add-one-1 '(3 1 5))

(add-one-2 '(3 1 5))

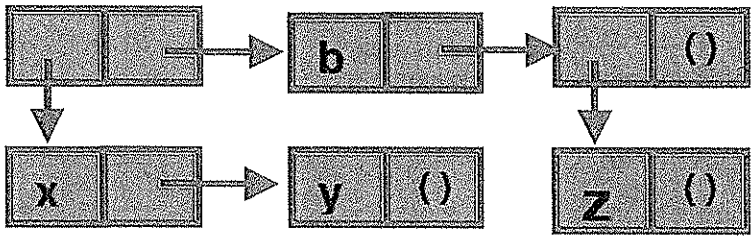
Uppgift 3. (2 poäng) Par/cons-celler

2a. (0,5 p) 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 (list (cons 'x '()) (list 'a)))

test    =>
```

2b. (0,5p) Skriv ett Scheme-uttryck som skapar följande struktur

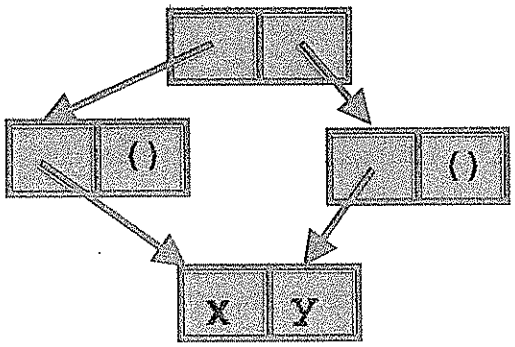


Uppgift 3. Par/cons-celler, forts

2c. (0,5p) Rita upp värdet med den grafiska representationen med cons-celler och pekare (*box and pointer notation*).

```
(let ((cells (cons 'a 'b)))
   (list cells cells))
```

2d. (0,5p) Skriv Scheme-uttryck som skapar nedanstående struktur:



Uppgift 4. (4 poäng) Rekursiva funktioner över listor

4a. (2p) Skriv en funktion (början? *start lista*), som undersöker om elementen på *start* är början av elementen på *lista*. Elementen kan testas med eq?.

```
(början? '(s t a r t) '(s t a r t a))     => #t
(början? '(s t a r t) '(s t a r t))       => #t      ; lika
(början? '(s t a r t) '(s t a r))         => #f      ; inte tillräckligt med element i list
(början? '(s t a r t) '(s t a r k a r e)) => #f
```

```
(define (början? start  lista)
```

Uppgift 4. Rekursiva funktioner över listor, forts

4b (2p) Skriv en funktion (max-nivå *lista*), som tar en godtycklig lista (en *proper list*, dvs som ej avslutas med punkterat par) och anger nivån för den lista som ligger på djupaste nivån. Toppnivån är 1.

```
(max-nivå '(a b c))          => 1, alla elementen är på toppnivån
(max-nivå '(a (b c) d e))    => 2 , andra elementet är en lista på nivå 2
(max-nivå '(a (b (c)) (d) e)) => 3 . listan (c) är djupast på nivå 3
```

Hjälp: Det finns en funktion max i Scheme, som returnerar den största parametern,
(max 2 3) = 3

```
(define (max-nivå lista)
```

Uppgift 5. (3 poäng) ADT – Abstrakt datatyp för aritmetiska formeluttryck

Vi vill kunna hantera aritmetiska formeluttryck (typen **aritm-uttryck**). Det aritmetiska uttrycket består av konstanter (typen **konstant**) och sammansatta uttryck (typen **uttryck**) med operatorerna + och – (typen **operator**) . Här skriver vi ett aritmetiska uttryck fullt parentetiserat.

4
(2 + 3)
((6 – 4) + 5)

Vi väljer en typad (*tagged*) representation i Scheme med typen först enligt följande:

(2 + 3) representeras som (uttryck (konstant 2) + (konstant 3))
((6 – 4) + 5) representeras som
(uttryck (uttryck (konstant 6) – (konstant 4)) + (konstant 5))

dvs konstanter typas med konstant och ett sammansatt uttryck med uttryck. Operatorerna typas ej.

Vi inför gränssnittsfunktioner för att skapa och ta ut delar ur aritmetiska formeluttryck.

```
(define (aritm-uttryck? obj)        ; godtyckligt-objekt -> sanningsvärde
   (or (konstant? obj) (uttryck? obj)))

(define (skapa-konstant tal)        ; heltal -> konstant
   (list 'konstant tal))

(define (konstant? obj)             ; godtyckligt-objekt -> sanningsvärde
   (eq? (car obj) 'konstant))

(define (värde konstant)            ; konstant -> heltal
   (car (cdr konstant)))

(define (skapa-uttryck operand1 operator operand2)
                    ; aritm-uttryck x operator x aritm-uttryck -> uttryck
  ... se uppgift 5a ...)

(define (uttryck? obj)              ; godtyckligt-objekt -> sanningsvärde
  ... se uppgift 5a ...)

(define (första-operand uttryck) ; uttryck -> aritm-uttryck
  ... se uppgift 5a ...)

(define (andra-operand uttryck)  ; uttryck -> aritm-uttryck
  ... se uppgift 5a ...)

(define (operator uttryck)          ; uttryck -> operator
  ... se uppgift 5a ...)
```

För att skapa de två första av ovanstående aritmetiska formeluttryck skriver vi:

```
(skapa-uttryck (skapa-konstant 2) '+ (skapa-konstant 3))

(define uttryck1
        (skapa-uttryck (skapa-uttryck (skapa-konstant 6) '- (skapa-konstant 4))
                       '+
                       (skapa-konstant 5)))
```

5a. (1p) Definiera följande gränsnittsfunktioner:

```
(define (skapa-uttryck operand1 operator operand2)
                                   ; aritm-uttryck x operator x aritm-uttryck -> uttryck

(define (uttryck? obj)             ; godtyckligt-objekt -> sanningsvärde

(define (första-operand uttr)      ; uttryck -> aritm-uttryck

(define (andra-operand uttr)       ; uttryck -> aritm-uttryck

(define (operator uttr)            ; uttryck -> operator
```

5b. (2p) Definiera en procedur beräkna, som tar ett aritmetiskt formeluttryck, t.ex. uttryck1 ovan, och räknar ut värdet av formeluttrycket.

(beräkna uttryck1) = dvs. $((6-4)+5) \Rightarrow 7$

```
(define (beräkna aritm)            ; aritm-uttryck -> heltal
```

Uppgift 6. (1 poäng) Högre ordningens funktion för binära träd

Vi har följande högre ordningens funktion (bearbeta-bt *bt löv-fn nod-fn*), som tar ett binärt träd (som cons-par) *bt* och två procedurer *löv-fn* och *nod-fn*. Proceduren *löv-fn* skall appliceras på varje löv och proceduren nod-fn skall applicera på de två värden, som returneras för resp. delträd i en nod.

```
(define (bearbeta-bt bt löv-proc nod-proc)
    (define (traversera bt)
        (if (löv? bt)
            (löv-proc bt)
            (nod-proc (traversera (vänster-bt bt))
                      (traversera (höger-bt bt)))))
    (traversera bt))

(define (löv? bt) (not (pair? bt)))
(define vänster-bt car)
(define höger-bt cdr)
```

a. Vad blir värdet av:

```
(bearbeta-bt '((1 . 2) . 3) (lambda (löv) (+ löv 10)) +)  =>
```

b. Vad blir värdet av:

```
(bearbeta-bt '((a . b) . c)
             (lambda (löv) (eq? 'b löv))
             (lambda (v1 v2) (and v1 v2)))  =>
```

c. Fyll i ? i nedanstående uttryck, så att vi givet ett binärt träd skapar en lista av alla löven.

```
(bearbeta-bt  '((a . b) . (c . (d . e))) ?a ?b) => (a b c d e)

?a =
?b =
```

d. Skriv en funktion (byt-plats-på-grenar *bt*), som med hjälp av bearbeta-bt, i ett binärt träd *bt*, skapar ett nytt binärt träd där vänster och höger delträd har bytt plats.

```
(byt-plats-på-grenar  '((1 . 2) . 3)) => (3 . (2 . 1)) = (3 2 .1)

(define (byt-plats-på-grenar bt)
```

Poäng på hela uppgiften:

0-1 rätta deluppgifter	0p
2- 3 rätta deluppgifter	0,5p
4 rätta deluppgifter	1.0p