Linköpings tekniska högskola
Institutionen för datavetenskap

Tentamen i

TDDC67 Funktionell programmering och Lisp

och äldre kurser

TDDC57 Programmering, Lisp och funktionell programmering

TDDC80 Programmering, Lisp och funktionell programmering

Tisdag den 18 december 2012 kl 8-13

Hjälpmedel: Inga.

Poänggränser: Maximalt kan erhållas 60p. För godkänt krävs ca 24p

Examinator: Peter Dalenius

Allmänna direktiv:

- Skriv läsligt och använd väl valda benämningar på funktioner och parametrar!
- Indentera programkoden enligt vedertagen praxis!
- Börja varje huvuduppgift på ett nytt blad!
- Läs igenom hela tentan! Det är inte säkert att de lättaste uppgifterna kommer först.
- Är du osäker på någon Common Lisp-funktion, beskriv hur du tror att den fungerar och vilka parametrar den tar. Om funktionen finns och gör ungefär det som du tror får du inga poängavdrag, även om parametrarna är i fel ordning.

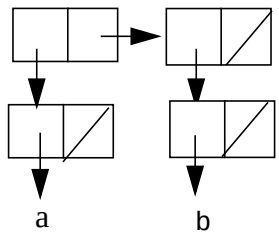
Lycka till!

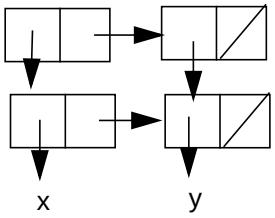
Uppgift 1. Listor, grafisk representation, evaluatorn (10 poäng)

1a. (1p) Rita upp grafiskt (med cons-celler och pilar) hur den liststruktur ser ut och vilket värde (i parentesformat) som skapas av

```
(cons 'x (list 'y (cons 'z '())))
```

1b. (2p) Skriv Lisp-uttryck som skapar nedanstående liststrukturer. Vilket värde i parentesformat skrivs ut av dessa liststrukturer.





1c. (3p) Rita upp grafiskt (med cons-celler och pilar) hur den liststruktur ser ut och vilket värde (i parentesformat) som skapas av

```
(setq x (list 'x 'y))
(setq y (list x 'x 'z)
(setf (rest x) (rest y))
```

Vilket värde har nu fölajnde?

```
x => ?
y => ?
```

1d. (4p) Vad blir värdet eller eventuellt fel när följande uttryck blir beräknat av Lisp-interpretatorn:

```
(funcall #'cons (+ 1 2) (+ 3 4))
(apply #'cons '((+ 1 2) (+ 3 4)))
(funcall #'(lambda (f g) (funcall g f f)) 'list #'list)
(eval (list 'list (list 'quote 'list)))
```

Uppgift 2. Olika modeller för att definiera funktioner (13 poäng)

2a. (3p) Skriv en *rekursiv funktion* snitt, som utför snittet mellan två mängder. Den rekursiva funktionen skall utföra en *rekursiv processlösning*. Mängderna är representerade som vanliga listor. Ordningen av elementen i resultatet har därför ingen betydelse.

```
(snitt '(a b c) '(x a y c))
=> (a c)
```

2b. (3p) Som 2a men den *rekursiva funktionen* skall utföra en *iterativ processlösning*.

2c. (3p) Som 2a men funktionen skall vara en *iterativ funktion*.

2d. (4p) Skriv en *destruktiv* variant av snitt och kalla den snitt!. Inga nya cons-celler får allokeras. Funktionen skall returnera som värde den nya listan. Ordningen på elementen är utan betydelse. Funktionen skall även klara av tomma mängder.

```
(setq a-lista '(a b c x))
(setq b-lista '(a x y c z))
(snitt! a-lista b-lista)
=> (a c x) eller annan ordning på elementen
```

Med din lösning vad har de globala variablerna a-lista och b-lista fått som nya värden?

Uppgift 3. Olika former av dubbelrekursion (10 poäng)

3a. (3p) Skriv en funktion ta-bort-alla-tal, som i en godtycklig lista (som dock ej innehåller punkterade par) tar bort alla talen. Övriga element skall returneras i en lista med samma struktur, som den givna.

```
(ta-bort-alla-tal '(3 (4 a) (5) (b (6 c))))
=> ((a) () (b (c)))
```

3b. (4p) Skriv en funktion postfix, som tar ett aritmetiskt uttryck med infix representation och som genererar ett postfix-uttryck, dvs ett uttryck där alla operander kommer före operatorerna. Infixuttrycket är representerat som en symbol eller nästlade listor, medan postfixuttrycket är representerat som en rak lista.

```
(postfix '3)
=> (3)

(postfix '(2 + 3))
=> (2 3 +)

(postfix '(1 + (2 * ((3 + 4) - 5))))
=> (1 2 3 4 + 5 - * +)
```

3c. (5p) I en associationslista har man lagrat personers barn. I exemplet nedan har linus barnen eva och per, eva i sin tur barnen emilia och emil.

```
(setq barntabell
'((linus . (eva per)) (linnea . (per)) (eva . (emilia emil))
  (per . (stina)) (stina . (lillan))))
```

Vi förutsätter att alla namnen är unika. Vi önskar en funktion avkomlingar som finner alla avkomlingar relativt en person. Ordningen mellan avkomlingarna är utan betydelse.

```
(avkomlingar 'linus barntabell)
=> (eva emilia emil per stina lillan)

(avkomlingar 'linnea barntabell)
=> (per stina lillan)

(avkomlingar 'torsten barntabell)
=> ()
```

Uppgift 4. Högre ordningens funktioner (6 poäng)

4a. (1p) Vad blir värdet av följande uttryck?

```
(mapcar #'second '((a b c) (1 2 3) (x y z)))
```

4b. (3p) Definiera en funktion analysera som tar en lista, som en rak sekvens samt en funktion och undersöker om elementen i sekvensen följer ett visst mönster. Mönstret specificeras genom funktionen som applicerat på ett element i sekvensen ska ge nästa element. En tom sekvens eller en sekvens med endast ett element uppfyller alltid mönstret. Exempel:

```
(analysera '(3 4 5 6 7) #'1+)
=> t        ; genom att addera ett element med1 erhålls nästa element

(analysera '(1 5 17 53 161) #'(lambda (x) (+ 2 (* 3 x))))
=> t        ; nästa element erhålls genom att multiplicera ett element med 3 och lägga till 2

(analysera '((a b c) (b c) (c) nil) #'rest)
=> t
```

4c. (2p) Definiera en lämplig funktion dubbla-udda?, som med hjälp av analysera undersöker en lista med tal. Om ett tal är jämnt skall det följas av nästa tal, är det udda skall nästa tal vara dubblerat.

```
(dubbla-udda? '(1 2 3 6 7 14 15 30 31 62 63))
=> t
```

Uppgift 5. Övriga konstruktioner (8 poäng)

5a. (4p) Vi har en arraystruktur med poster. Varje post består av två fält, ett namn och en ålder. Vi önskar en funktion del-av-array som givet två namn skapar en ny arraystruktur med alla posterna som ligger mellan (inklusive) dessa två namn i den ursprungliga tabellen. Dessutom skall åldrarna ökas med 1.

Posterna skall vara av datatypen student med fältnamnen namn och ålder. Definiera denna datatyp och funktionen del-av-array enligt nedanstående exempel, där vi skapar en arraystruktur med 6 postelement:

```
(del-av-array
   (make-array 6 :initial-contents
      (list (make-student :namn 'kalle :ålder 17)
            (make-student :namn 'anna :ålder 18)
            (make-student :namn 'lisa :ålder 15)
            (make-student :namn 'karin :ålder 22)
            (make-student :namn 'stina :ålder 13)
            (make-student :namn 'mia :ålder 10)))
   'anna
   'karin)
=> #(#S(student :namn anna :ålder 19)
     #S(student :namn lisa :ålder 16)
      #S(student :namn karin :ålder 23)), dvs en arraystruktur med 3 postelement
```

Eftersom man inte på förhand vet antalet element som skall lagras i den nya arraystrukturen är det tillåtet att först samla ihop dessa element i en lista för att senare lagras som element i den nya arraystrukturen.

5b. (4p) Vi önskar en konstruktion i Common Lisp repetera, som tar ett uttryck (en form) och ett heltal *n*, och beräknar uttrycket *n* gånger. Man kan tänka sig vilja använda konstruktionen vid tidtagning, så att man kan beräkna ett enkelt uttryck ett stort antal gånger. Andra argumentet skall först beräknas och endast en gång. Definiera repetera. Värdet är ej av intresse.

```
(repetera (cons nil nil) 100000), skapar 100.000 cons-celler

(setq nr 0)
(repetera (progn (setq nr (1+ nr)) (print nr)) (* 10 10)), skriver ut talen 1 till 100
```

Uppgift 6: Abstraktion (13p)

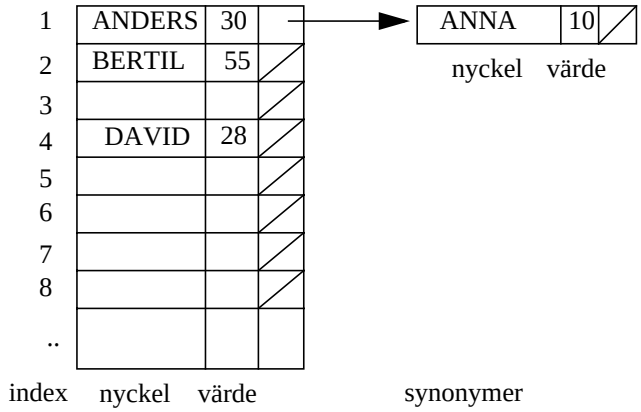
En *hashtabell* är organiserad så att man snabbt skall kunna lagra och hitta element i tabellen. Givet att man har en tabell av en fix storlek, säg plats för 100 element. Hela elementet eller del därav har man som *nyckel*. Nyckeln används för att finna plats där elementet skall lagras eller redan finns. En vanlig nyckel i alla databassammanhang är ju vårt omdebatterade personnummer. Vanligen kan inte nyckeln direkt användas utan man måste transformera den till ett *index* i tabellen. För transformationen har man en *hashfunktion*. Om tabellen är 100 element stor skall nycken transformeras till ett värde mellan 1 och 100. Detta medför emellertid att man kan få samma index för olika nycklar och måste därför ha en sätt att hantera *synonymer*. Elementen i själva hastabellen brukar kallas för *buckets*.
Ett sätt att organisera detta är att ha en direktaccess-struktur, dvs en ändlig avbildning mellan index och en "rad" i tabellen och en möjlighet att för varje index ha en länkad lista för att lagra synonymerna.

Exempel: Tabell av storlek 28. Nyckel är vanliga namn. Med nyckeln associeras en ålder. Hashfunktion är ordningsnumret i alfabetet av första bokstaven i namnet, dvs ANDERS får index 1, BERTIL index 2, men även ANNA får index 1, som då kommer att lagras i synonymlistan.

Vi lagrar:

ANDERS	30
BERTIL	55
ANNA	10
DAVID	28

Tabellen kan då grafiskt beskrivas som:



Vi inför följande abstrakta datatyper:

hashtable [bucket, ..., bucket], dvs som en *ändlig avbildning* med element av typen **bucket** och med indexmängden **index**, för att ange en hashtabell

index är en primitiv typ, för att ange index för objekt av typen **hashtable**

bucket <**contents**, **synonymlist**>, dvs som en *tupel* av **contents** och **synonymlis**t samt värdet nil för att ange att nyckel-värde-par saknas för ett givet index**.**

contents <**key**, **keyvalue**>, dvs som en *tupel* av **key** och **keyvalue**, för att ange nyckel och värde

key är en primitiv typ, som anger nyckel

keyvalue är en primitiv typ, som anger värdet

synonymlist{{**contents**}}*, dvs som en *sekvens* av element av typen **contents**, för att ange synonymer

I bilagana finns funktioner för hantering av en hashtabell.

Vi har en huvudoperation:

insert-hashtable som lägger in en nyckel och ett värde i en hashtabell. Finns redan nyckeln i tabellen görs ingenting (även om nyckelvärdet skulle skilja).

Uppgifter:
Läs och förstå koden i bilagan.

6a. (4p) Välj representationen enligt nedan för att representera objekten av datatypen **hashtable**. Definiera primitiverna make-hashtable, insert-bucket och get-bucket.
Representation: Associationslista med par med index och ett objekt av typen **bucket**, dvs på formen
((1 . *värde-av-typ-bucket*) (2 . *värde-av-typ-bucket*) .... (n . *värde-av-typ-bucket*))

6b. (4p) Funktionen in-synonymlist? saknas. Definiera den. Funktionen testar om ett element med samma nyckel tidigare lagrats på synonymlistan, i så fall skall ett sant värde returneras.

6c. (5p) Skriv en funktion delete-hashtable, med nyckel och hashtabell som argument, som tar bort nyckel och värdet ur tabellen. Om elementet i själva hashtabellen tas bort och det finns synonymer skall en av synonymerna läggas över i själva tabellen. Nya primitiva funktioner få införas. Finns redan lämplig primitiv skall den användas.

Bilaga. Funktioner för uppgift 6

Funktioner för hantering av en hashtabell:

```
(defun insert-hashtable (new-entry new-val ht)
       ;; key x keyvalue x hashtable -> hashtable
     (let ((index (hashfn new-entry ht)))
       (cond
           ((in-basic-hashtable? new-entry (get-bucket ht index)) ht)
           ((empty-bucket? (get-bucket ht index))
               (insert-bucket
                   ht
                   index
                   (make-bucket
                      (make-contents new-entry new-val)
                      (init-synonymlist))))
            ((in-synonymlist? new-entry (synonymlist (get-bucket ht index)))
                ht)
           (t (insert-bucket
                   ht
                   index
                   (make-bucket
                     (contents (get-bucket ht index))
                     (extend-synonymlist
                        (make-contents new-entry new-val)
                        (synonymlist (get-bucket ht index)))))) )))

(defun hashfn (entry ht)
       ;; key x hashtable -> index
     .. den exakta definitionen behövs ej ..)

(defun in-basic-hashtable? (entry bk)
       ;; key x bucket -> boolean
    (eql entry (get-entry (contents bk))))

(defun in-synonymlist? (entry slist)
       ;; key x synonymlist -> boolean
     ... se uppgift 6b ...)
```

Primitiva funktioner för datatypen **hashtable**

```
(defun make-hashtable (size)
       ;; integer -> hashtable
     .. se uppgift 6a ..))

(defun insert-bucket (ht index bk)
       ;; hashtable x index x bucket -> hashtable
     .. se uppgift 6a..)

(defun get-bucket (ht index)
       ;; hashtable x index -> bucket
    .. se uppgift 6a ..)
```

Primitiver för datatypen **bucket**

```
(defun make-bucket (cont slist)
       ;; contents x synonymlist -> bucket
    (cons cont slist))
```

```
(defun contents (bk)
       ;; bucket -> contents
    (first bk))

(defun synonymlist (bk)
       ;; bucket -> synonymlist
    (rest bk))

(defun empty-bucket? (bk)
       ;; bucket -> boolean
    (eq bk 'nil))
```

Primitiver för datatypen **contents**

```
(defun make-contents (entry val)
       ;; key x keyvalue -> contents
    (cons entry val))

(defun get-entry (cont)
       ;; contents -> key
     (first cont))

(defun get-value (cont)
       ;; contents -> keyvalue
    (rest cont))
```

Primitiver för datatypen **synonymlist**

```
(defun init-synonymlist ()
       ;; -> synonymlist
    '())

(defun empty-synonymlist? (slist)
       ;; synonymlist -> boolean
    (endp slist))

(defun extend-synonymlist (cont slist)
       ;; contents x synonymlist -> synonymlist
    (cons cont slist))

(defun first-contents (slist)
       ;; synonymlist -> contents
     (first slist))

(defun rest-synonymlist (slist)
       ;; synonymlist -> synonymlist
     (rest slist))
```

Hjälpfunktion i för att hantera associationslistor.

```
(defun make-alist (size)
      ;; returnerar en associationslista ((1 . nil) .. (size . nil))
   (labels ((make-alist-size (nr size)
              (cond
                 ((> nr size) nil)
                 (t (cons (cons nr nil) (make-alist-size (1+ nr) size))))))
 (make-alist-size 1 size))
```