Linköpings universitet
Institutionen för datavetenskap
Peter Dalenius

Tentamen i

TDDC67 Funktionell programmering och Lisp

Även vid behov tentamen för följande avslutade kurser:
TDDC80, TDDC57, TDDB81, TDDB80, TDDB92, TDDB93

Onsdag 27 mars 2013 kl 14-19

Hjälpmedel: Inga.

Poänggränser: Maximalt kan erhållas 55 poäng. För godkänt krävs cirka 22 poäng.

Jourhavande lärare: Peter Dalenius, telefon 013-28 14 27

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!

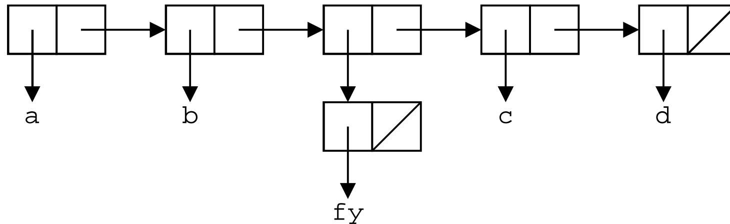
Observera att denna tentamen – utöver detta tillfälle – endast ges ytterligare två gånger:
21 augusti 2013 8-13 samt 9 januari 2014 8-13.

Uppgift 1 (10p)

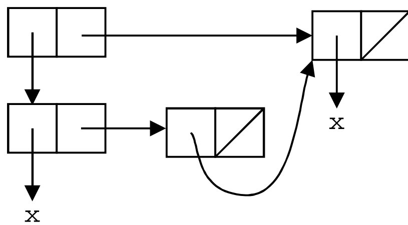
1a. (1p) Rita den grafiska representationen av den liststruktur som skapas av följande uttryck:

```
(cons ’a (list ’(b) ’()))
```

1b. (1p) Skriv ett Lisp-uttryck som skapar följande liststruktur:



1c. (4p) Vi har skapat nedanstående liststruktur:



Vilket eller vilka av nedanstående uttryck motsvarar ovanstående liststruktur, d.v.s. hur kommer liststrukturen att skrivas ut av Lisp-systemet?

```
A: ((x x) x)
B: ((x (x)) (x))
C: ((x (x)) x)
D: (((x) x) x)
```

Vilket eller vilka av nedanstående uttryck skapar ovanstående liststruktur?

```
E: (list (list ’x (list ’x)) ’x)
F: (let ((y (list (list ’x))))
     (cons (cons ’x y) (first y)))
G: (let ((y (cons ’x ’())))
     (cons (list ’x y) ’(x)))
H: (let ((y (list (list ’x ’(x)))))
     (setf (rest y) (second (first y)))
     y)
```

För att svara ska du rita en tabell med alternativ A till H och för vart och ett av dem ange JA eller NEJ om alternativet är korrekt eller inte. Varje rätt svar ger +0,5p, varje felaktigt svar ger -0,5p och ett uteblivet svar ger 0p. Denna deluppgift kan dock aldrig ge mindre än 0p totalt.

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

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

Uppgift 2 (15 p)

2a. (4p) Skriv en funktion `anagram?` som testar om en rak lista av symboler är ett anagram av en annan lista, d.v.s. den innehåller exakt samma symboler men inte nödvändigtvis i samma ordning. Du får gärna dela upp lösningen i flera funktioner om du känner att det är nödvändigt, men minst en funktion måste vara *rekursiv*. Ange för dina rekursiva funktioner om de uppvisar en *rekursiv* eller *iterativ processlösning*. Motivera varför det är så. Exempel:

```
CL-USER(1): (anagram? '(p e t e r) '(r e e p t))
T
CL-USER(2): (anagram? '(p e t e r) '(r e e p q))
NIL
CL-USER(3): (anagram? '(k a l l e) '(l e k a))
NIL
CL-USER(4): (anagram? '() '())
T
```

2b. (3p) Skriv en *rekursiv* funktion `mergesort` som samsorterar två raka sorterade listor med tal, d.v.s. resultatet ska vara en ny lista som också är sorterad. Funktionen ska utnyttja egenskapen att listorna är sorterade. Listorna är sorterade i stigande ordning. Exempel:

```
CL-USER(5): (mergesort '(1 3 5 7) '(2 3 6 8))
(1 2 3 3 5 6 7 8)
CL-USER(6): (mergesort '() '(1 2 3))
(1 2 3)
```

2c. (2p) Skriv en *iterativ* version av funktionen `mergesort` från föregående uppgift.

2d. (3p) Skriv en funktion `max-djup` som undersöker hur djupt ett binärt träd är. Det binära trädet representeras som punkterade par. Ett löv har djupet 0. Exempel:

```
CL-USER(7): (max-djup 'a)
0
CL-USER(8): (max-djup '(a . b))
1
CL-USER(9): (max-djup '((a . (b . c)) . d))
3
```

2e. (3p) Skriv en funktion `nivåer` som givet en godtycklig lista (utan punkterade par) returnerar en lista med nummer på de nivåer där en given symbol finns. Toppnivån har djupet 1. Om symbolen inte finns alls returneras tomma listan. Ordningen på nivånumren i resultatlistan ska komma i den ordning elementen finns i listan. Exempel:

```
CL-USER(10): (nivåer 'x '(a (b) x))
(1)
CL-USER(11): (nivåer 'x '(a (b) c))
NIL
CL-USER(12): (nivåer 'x '((a (x)) x x (b c)))
(3 1 1)
CL-USER(13): (nivåer 'x '((a x) x (b x)))
(2 1 2)
```

Uppgift 3 (10p)

3a. (4p) Vi har en arraystruktur som innehåller 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 poster från och med det första namnet till och med det sista. Posterna ska vara av datatypen `student` med fältnamnen `namn` och `ålder`. Definiera denna datatyp och funktionen `del-av-array` enligt nedanstående exempel:

```
CL-USER(1): (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 'muhammed :ålder 13)
                      (make-student :namn 'mia :ålder 10)))
              'anna
              'karin)
#(#S(student :namn anna :ålder 18)
  #S(student :namn lisa :ålder 15)
  #S(student :namn karin :ålder 22))
```

Eftersom man inte på förhand vet antalet element som ska läggas in i den nya arraystrukturen är det tillåtet att först samla ihop dessa element i en lista för att senare överföra dem till en arraystruktur. Dessutom antar vi att namnen alltid finns och förekommer i rätt ordning i arrayen. För denna uppgift får du poängavdrag om du inte kan namnen på de funktioner som opererar på arrayer och poster.

3b. (3p) I många andra språk, t.ex. C++ och Java, finns en konstruktion `++` som ökar en variabels värde med ett. Man kan då skriva antingen `++i` eller `i++`. Definiera en sådan operation i Common Lisp och kalla den `op++`. Värdet av operationen ska vara det nya värdet. Exempel:

```
CL-USER(1): (setq n 10)
10
CL-USER(2): (op++ n)
11
CL-USER(3): n
11
CL-USER(4): (progn (op++ n) (op++ n) 'ok)
OK
CL-USER(5): n
13
```

3c. (3p) Vi definierar följande funktion och evaluerar `let`-uttrycket nedan:

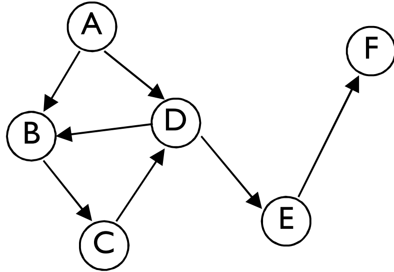
```
(defun f2 (init)
  #'(lambda (n) (setq init (+ n init)) init))

(let ((a (f2 10)) (b (f2 20)))
  (mapcar #'(lambda (f n) (funcall f n))
          (list a b a b)
          '(1 3 5 7)))
```

Förklara vad som returneras från funktionen `f2`. Förklara vad `let`-uttrycket returnerar och motivera detta.

Uppgift 4 (10p)

4a. (6p) Vi har givet en graf och en startnod i grafen. Vi vill undersöka om man utgående från startnoden kan hamna i loop i grafen. Exempel:



Med startnoden A kan vi hamna i en loop om vi går A-B-C-D-B, d.v.s vi har en loop B-C-D. Med startnoden E kan vi ej hamna i en loop eftersom vi enbart kan gå vidare till F.

Skriv en funktion `loop?` som tar en graf och en startnod och som returnerar ett sanningsvärde som representerar om det fanns en loop eller ej.

Grafens representation behöver du ej fundera över. Antag att det finns en primitiv funktion barn som givet en nod och en graf ger en lista med nodens efterföljander. Noderna betecknas med vanliga symboler. Om vi antar att vi har ovanstående graf lagrad i variablen `test-graf` kan vi alltså skriva:

```
CL-USER(1): (barn ’d test-graf)
(B E)
CL-USER(2): (barn ’f test-graf)
NIL
CL-USER(3): (loop? ’a test-graf)
T
CL-USER(4): (loop? ’c test-graf)
T
CL-USER(5): (loop? ’e test-graf)
NIL
```

4b. (4p) Förklara hur konstruktionerna `catch` och `throw` fungerar. Ge ett illustrativt exempel där du använder dessa för att hantera fel.

Uppgift 5 (10p)

Vi önskar analysera Lisp-program och vill därför ha en funktion `code-analyze` som går igenom ett Lisp-uttryck på ett korrekt sätt och ger oss viss information. Ett Lisp-uttryck kan (i den här uppgiften) vara ett av följande:

- konstanter, t.ex. 4711, 3.14
- sanningsvärden, `t` eller `nil`
- quote:ade uttryck, t.ex. `(quote kalle)`
- variabler, d.v.s. symboler
- villkorliga uttryck i form av `cond`-uttryck på formen
```
(cond
  (uttryck uttryck)
  (uttryck uttryck)
  ...
  (uttryck uttryck))
```
- funktionsapplicering på formen
```
(funktionsnamn uttryck uttryck ... uttryck)
```
- definition av lokala variabler med let på formen
```
(let ((variabel uttryck) (variabel uttryck) ... (variabel uttryck))
  uttryck)
```

Funktionen `code-analyze` ska ta ett Lisp-uttryck, analysera det och skapa en informationslista med följande innehåll

```
(variabellista funktionslista)
```

d.v.s. vi får fram en lista med alla variabler som används i uttrycket och en lista med alla funktioner som anropas från uttrycket. Exempel:

```
CL-USER(1): (code-analyze '(+ 3 x))
((X) (+))
CL-USER(2): (code-analyze '(let ((x (+ 10 z))) (+ x y)))
((X Y Z) (+))
CL-USER(3): (code-analyze '(cond (flg t) (t (first l))))
((FLG L) (FIRST))
CL-USER(4): (code-analyze '(cons (g x) (cons (car l) (h y))))
((X L Y) (CONS G CAR H))
CL-USER(5): (code-analyze '(cons f (quote (car cdr))))
((F) (CONS))
```

Vi vill att vårt analysprogram ska definieras på en abstrakt nivå. Därför definierar vi en abstrakt datatyp **info** som hanterar den informationslista som vi vill bygga upp. Längre ner i uppgiften definieras några primitiver:

`empty-info` skapar den tomma informationslistan
`variable-info` skapar en initial informationslista med information om en variabel
`function-info` skapar en initial informationslista med information om en funktion
`merge-info` slår ihop informationen i två informationslistor

På nästa sida ges själva uppgifterna.

5a. (3p) Gör en abstraktion över de Lisp-uttryck som listades ovan. Inför abstrakta datatyper och primitiver för dessa. Du behöver bara namnge primitiverna och ange datatyper, inte implementera dem fullständigt.

5b. (7p) Definiera funktionen `code-analyze` med hjälp av de primitiver du definierade i föregående uppgift samt primitiverna för datatypen **info** nedan.

```
;;; Primitiver för datatypen info --------------------

(defun empty-info ()             ; -> info
  (list '() '()))

(defun variable-info (var)       ; variable-name -> info
  (list (list var) '()))

(defun function-info (fun)       ; function-name -> info
  (list '() (list fun)))

(defun merge-info (info1 info2)  ; info x info -> info
  (list
    (union (first info1) (first info2))
    (union (second info1) (second info2))))
```