Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2014-03-26
Sal (2) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	(TER1) TERD
Tid	14-16
Kurskod	TDDC74
Provkod	TEN1
Kursnamn/benämning **Provnamn/benämning**	Programmering - abstraktion och modellering Skriftlig tentamen/duggor
Institution	IDA
Antal uppgifter som ingår i tentamen	3
Jour/Kursansvarig Ange vem som besöker salen	Johannes Schmidt
Telefon under skrivtiden	0725-721800
Besöker salen ca kl.	ja (ca 08:30)
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	

TDDC74 Programmering: Abstraktion och modellering
Dugga 3, kl 14—16, 26 mars 2014

Läs alla frågorna först, och bestäm dig för i vilken ordning du vill lösa uppgifterna.

Skriv tydligt och läsligt. Använd **väl valda namn** på parametrar och **indentera** din kod.

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

OBS! Denna dugga får du använda kommandon som `set!`, `set-mcar!`, `set-mcdr!` för att ändra bindningar och ändra i strukturer.

Betyg: Det finns tre duggor i kursen. På varje dugga kan man få som mest 12p (så totalt 36p för alla tre).

För att bli godkänd på en dugga krävs minst 3p.

För att kunna få betyget 3 på duggorna måste du sammanlagt (på alla tre duggor) ha fått minst 18p, för betyget 4 gäller minst 23p och för betyget 5 minst 27p.

För att bli godkänd på duggaserien måste du dessutom ha godkänt på samtliga duggor.

Lycka till!

Uppgift 1 (4 poäng)

a) **(2 poäng)** Implementera proceduren `maintain-max`. Denna ska ta ett argument, ett positivt heltal, och returnera det största positiva heltal som den hittills har blivit anropad med.

Så här ska den fungera:

```
> (maintain-max 2)
2
> (maintain-max 7)
7
> (maintain-max 1)
7
> (maintain-max 6)
7
```

b) **(2 poäng)** Rita ett omgivningsdiagram som visar vad som hänt när du har evaluerat såväl definitionen av `maintain-max`, som uttrycken ovan. Markera globala omgivningen (GE) tydligt, och numrera ramarna i den ordning de skapas (E1, E2,...).

Du behöver inte skriva ut all kod som finns i procedurkroppar (body), men det ska tydligt framgå vilken kod du hänvisar till.

Uppgift 2 (4 poäng)

Nedan följer en konstruktor för ett lampobjekt.

```
(define (make-light init-color)
  (let ((color init-color))
    (lambda msg
      (cond
        ((null? msg) (printf "I am a ~a light.\n" color))
        ((eq? (car msg) 'get-color) color)
        ((eq? (car msg) 'reset-color!) (set! color init-color))
        ((eq? (car msg) 'set-color!) (set! color (cadr msg)))
        (else
         (printf "Unknown command: ~a\n" (car msg)))))))
```

a) (**1 poäng**) Skriv om koden ovan utan syntaktiskt socker, så att man lättare kan rita omgivningsdiagram. Gör sådana omskrivningar som presenterats vid föreläsningarna (exempelvis för `define`- och `let`-uttryck).

`cond`-uttrycket påverkar inte omgivningsdiagram, och behöver inte utvecklas. Du behöver inte heller skriva ut *hela* `cond`-uttrycket, men var noga med indenteringen, och hur du öppnar och sluter parenteser.

b) (**3 poäng**) Vi evaluerar definitionen ovan, och sedan följande uttryck:

```
(define mylight (make-light 'red))
(mylight 'get-color)
(mylight 'set-color! 'blue)
(mylight 'reset-color!)
```

Rita ett omgivningsdiagram som visar vad som hänt när du har evaluerat såväl definitionen av `make-light`, som uttrycken ovan. Markera globala omgivningen (GE) tydligt, och numrera ramarna i den ordning de skapas (E1, E2,...).

Du behöver inte skriva ut all kod som finns i procedurkroppar (body), men det ska tydligt framgå vilken kod du hänvisar till.

Uppgift 3 (4 poäng)

Vi definierar ett slags databasobjekt som innehåller kopplingar mellan nycklar och värden. För enkelhets skull är alla nycklar heltal, och alla värden symboler. Inuti objekten (som beskrivs nedan) representeras allt som en **muterbar lista** innehållande **muterbara par**.
Databasobjekten ska ha följande metoder:

1. `get-content`: returnerar hela databasens innehåll (hela listan)
2. `get-value`: tar fram värdet som associeras med en viss nyckel (om nyckeln finns i databasen)
3. `insert-pair!`: sätter in ett nytt muterbart par (nyckel-värde) i listan
4. `change-value!`: ändrar värdet som hör till en given nyckel (om nyckeln finns i databasen).

Ett exempel på objekten i användning följer på nästa sida.

Koden nedan visar konstruktorn för våra databasobjekt:

```
(define (make-database)
  (let ((content '()))
    (lambda msg
      (cond
        ((null? msg) (printf "I am a database.\n"))
        ((eq? (car msg) 'get-content) content)

        ((eq? (car msg) 'get-value)
         (let ((answer (massq (cadr msg) content)))
           (if (eq? answer #f)
               #f
               (mcdr answer))))

        ((eq? (car msg) 'insert-pair!)
         ...)

        ((eq? (car msg) 'change-value!)
         ...)

        (else
         (printf "Unknown command: ~a\n" (car msg)))))))
```

Uppgift Implementera `insert-pair!` och `change-value!`. Det finns ett antal särkilda procedurer som verkar på muterbara par. Följande är tillåtna (men *behöver inte* användas): `set-mcar!`, `set-mcdr!`, `massq`, `mappend`.

Här ser du hur våra databasobjekt ska fungera:

```
> (define mydb (make-database))
> (mydb 'get-content)
()

> (mydb 'insert-pair (mcons 12 'apple))
> (mydb 'insert-pair (mcons 4 'cow))
> (mydb 'insert-pair (mcons 3 'car))
> (mydb 'get-content)
{{3 . car} {4 . cow} {12 . apple}}

> (mydb 'get-value 12)
apple
> (mydb 'get-value 11)
#f

> (mydb 'change-value 12 'foo)
> (mydb 'get-value 12)
foo
> (mydb 'change-value 11 'bar)
#f
> (mydb 'get-content)
{{3 . car} {4 . cow} {12 . foo}}
```

OBS!

- Ordningen på elementen i databasen spelar ingen roll ({{3 . car}, {4 . cow},...}} eller {{4 . cow}, {3 . car},...}}, ...). Varje nyckel-värde-par ska däremot såklart vara i rätt ordning.
- Du kan anta att användaren aldrig försöker sätta in ett par med en redan existerande nyckel.

Tips

1. Proceduren `massq` fungerar precis som `assq`, men för muterbara par. Den tar två argument: först en nyckel, därefter en muterbar (associations-)lista som den ovan. Om något element i listan matchar nyckeln, returneras hela paret ({ `nyckel . värde` }). Annars returneras `#f`.
2. Tänk noga på när/om du vill använda `set!` respektive `set-mcar!/set-mcdr!`.