Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2017-10-27
Sal (1)	SU-salar, IDA(32)
Tid	14-18
Kurskod	TDDC74
Provkod	DAT1
Kursnamn/benämning **Provnamn/benämning**	Programmering - abstraktion och modellering Datortentamen
Institution	IDA
Antal uppgifter som ingår i tentamen	6
Jour/Kursansvarig Ange vem som besöker salen	Jalal Maleki och Anders Märak Leffler
Telefon under skrivtiden	Jalal (ankn. 1963), Anders (073-1011291)
Besöker salen ca klockan	ja (finns i närheten större delen av tiden)
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, anna.grabska.eklund@liu.se, ankn. 2362
Tillåtna hjälpmedel	Inga
Övrigt	
Antal exemplar i påsen	

TDDC74 Programmering: Abstraktion och modellering
Datortenta - 2017-10-27, kl 14-18

Läs alla frågorna först och bestäm dig för i vilken ordning du vill lösa uppgifterna. Uppgifterna är inte nödvändigtvis i svårighetsordning.

Använd **väl valda namn** på parametrar och **indentera** din kod. Väl valda namn inkluderar bland annat konsekvent språk. Du behöver inte skriva kodkommentarer, annat än för väldigt svårförklarade hjälpfunktioner (som bör undvikas). Namngivning ska vara tillräcklig (och följa konventioner). Skriv inte onödigt komplicerade lösningar.

Om det är naturligt, definiera gärna hjälpfunktioner! Hjälpfunktioner ska som vanligt lösa tydliga och lättförklarade uppgifter.

Du får använda **alla tillgängliga primitiver** och språkkonstruktioner i Racket, **om annat inte anges** i uppgiftens text.

Frågor

Är något otydligt i uppgifterna kan du använda meddelande-funktionen i tentaklienten för att skicka frågor till jourlärare.

Att lämna in

Döp dina filer till uppg1.rkt, uppg2.rkt, ... Filändelsen är relevant.

Skicka in uppgifterna med hjälp av tentaklienten, när du är klar med dem! Vänta *inte* på att alla uppgifter är klara med att lämna in. När du har lämnat in en uppgift, fortsätt arbeta på nästa. Du har en inlämning per uppgift (så skicka inte ev a- och b-uppgifter separat). **Följ angivna namn, och testa att alla körexempel i uppgiften fungerar exakt som de är inskrivna!**

Betyg: För trea räcker ca 50% av det totala poängen. För en fyra räcker ca 65% och för en femma ca 80%.

Ett resultat man inte är nöjd med, kan plussas vid ett senare tentamenstillfälle.

Lycka till!

Uppgift 1, Teori och semantik (3 poäng)

Lämna in dina svar som kommentar i fil `uppg1.rkt`.

1. Antag att vi har följande kod:

```
(define (return-if-non-zero n val)
  (if (zero? n)
   0
   val))

(define (slowpoke)
   ;; some calculation that takes 24h to complete
   999)
```

   Vi evaluerar uttrycket `(return-if-non-zero 0 (slowpoke))`. När (om) det hela är slut, får vi tillbaka siffran noll (eftersom $n = 0$). Kan vi säga från början att evalueringen av det uttrycket kommer att ta ca 24h? Varför? Svara kortfattat (max en-två meningar).[1](1p)

2. Vi startar DrRacket, skriver in koden nedan (utan att ha skrivit något annat först) och försöker köra den. Det fungerar inte. Varför inte? Svara kortfattat (max en-två meningar). (1p)

```
(define n (+ n 1))
```

3. Vissa språk, bland annat Racket, kan optimera funktioner som ger upphov till en iterativ process. Funktionen nedan är antingen linjärrekursiv eller iterativt rekursiv. Vilken av dem? Varför? Svara kortfattat (max en-två meningar). (1p)

```
(define (f1 n)
  (if (= n 0)
   112
   (f1 (- n 1))))
(f1 100)
```

[1] Skriv gärna testfall för att övertyga dig själv.

Uppgift 2, Enkelrekursion, högre ordningens procedur (5 poäng)

Lämna in dina svar i fil `uppg2.rkt`.

I matematiken kan vi se en funktion $f : X \rightarrow Y$ som någon form av regel som knyter ihop varje element i X med exakt ett element i Y.. Ett exempel på en funktion kan - lite slarvigt - vara "f(1) = 10, f(2) = 10, f(17) = 85" (och 1, 2, 17 är de enda tillåtna indata). Vi skulle kunna beskriva denna ändliga funktion genom "indata: `(1 2 17)`, utdata (i samma ordning): `(10 10 85)`".[2].

a) Din uppgift är att skriva en funktion `eval-fun` som tar en lista med möjliga indata, en lista med motsvarande utdata, och en punkt x där vi vill ta reda på funktionsvärdet $f(x)$. Första elementet i indata-listan hör ihop med första elementet i utdata-listan, och så vidare.

   Du kan anta att listorna är lika långa, ändliga, och att alla olika indatavärden är unika. Om x inte finns med bland tillåtna indata ska funktionen returnera strängen "undefined". Värdena ska jämföras med `eqv?`. Så här ska det fungera:

```
> (eval-fun '(1 2 17) '(10 10 85) 2)
10
> (eval-fun '(1 2 17) '(10 10 85) 17)
85
> (eval-fun '(1 2 17) '(10 10 85) 999) ;; 999 ej bland tillåtna indata!
"undefined"
> (eval-fun '(9 3 100) '(hunden katten glassen) 3) ;; valfri ordning av indata
'katten
```

b) Nu vill vi generera Racket-funktioner utifrån en funktionsbeskrivning som den ovan. Skriv en funktion `make-procedure` som tar en in- och utdatalista, och returnerar en funktion av en variabel x som returnerar funktionens värde i x. Så här ska den fungera:

```
> (define f (make-procedure '(1 2 17) '(10 10 85))) ;; in- och utdata som ovan
> f
#<procedure>
> (f 2)
10
> (f 999)
"undefined"
```

   OBS! Du får anropa din kod från a-uppgiften. **Har du inte svarat på a-uppg., antag att en sådan funktion finns.**

[2]Vi gör vissa förenklingar här; exempelvis struntar vi i vad målmängden Y är. Vi antar också att X är ändlig. Det är OK.

Uppgift 3. Dubbelrekursion, generella mönster (6p)

Lämna in dina svar i fil `uppg3.rkt`.

Skriv en procedur `replace-all` som tar en liststruktur `seq` av godtyckligt djup, och ersätter alla förekomster av symbolen `old` med symbolen `new`. `seq` kan innehålla såväl symboler som underlistor (som kan innehålla nya listor, och så vidare).

Du kan anta att alla listor är äkta listor, som uppfyller predikatet `list?`. [3]

Så här ska det fungera:

```
> (replace-all '(a b c d e) 'a 'foo)
'(foo b c d e)
> (define deep '(a dog (((((((dog fish)) dog)))) dog) 5))
> (replace-all deep 'dog 'cat)
'(a cat (((((((cat fish)) cat)))) cat) 5)
> (replace-all deep 999 'snake)  ;; no 999 in deep
'(a dog (((((((dog fish)) dog)))) dog) 5)
```

a) Redovisa kod för `replace-all`.

   Koden i a-uppgiften får inte innehålla anrop till `map`. (3p)

b) En stor del av koden du skrivit ovan utför mönstret "för varje element, gör *något* och samla upp resultatet". Just den biten är precis vad den klassiska inbyggda funktionen `map` gör.

   Skriv en funktion `ram` som fungerar exakt som `replace-all`, men är skriven på formen

```
(define (ram seq old new)
  (map
   ... din kod ...
   seq))
```

   Självklart får `ram` inte anropa `replace-all` (eller liknande).

   Du behöver inte skriva hela "... din kod ..." på en rad!

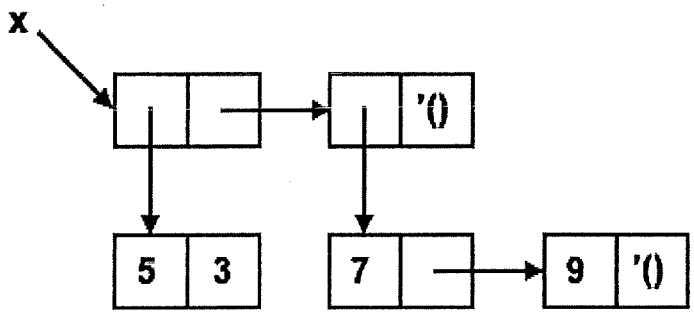
   Redovisa kod för `ram`. (3p)

[3] Alltså inga par som `'(head . tail)`.

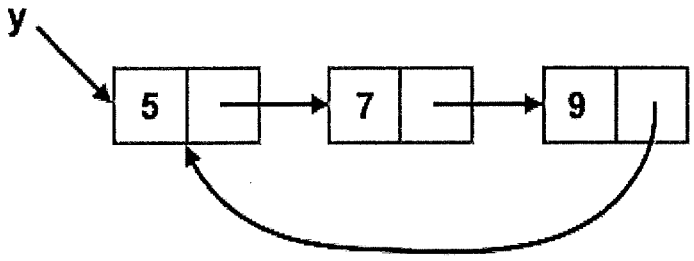
Uppgift 4, Liststrukturer, grafisk representation (3 poäng)

Lämna in dina svar i fil `uppg4.rkt`.

a) (1,5p) Skriv Schemeuttryck som skapar följande struktur. Du får inte använda `set-mcar!` och `set-mcdr!` för denna deluppgift eftersom de skall inte behövas. Använd gärna `define`, `let` och/eller `let*` för att tillfälligt benämna delstrukturer som du skall använda flera gånger.



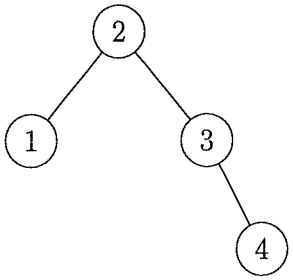
b) (1,5p) Skriv Schemeuttryck som skapar följande struktur. Du får använda `set-mcar!` och `set-mcdr!` vid behov. Använd gärna `define`, `let` och/eller `let*` även här.



Uppgift 5. Abstrakt datatyp, högre ordningens procedur (6p)

Vi vill implementera en dataabstraktion som kallas för binärträd. Ett binärträd kan här vara ett tomt binärträd eller en struktur som består av tre delar: ett värde, samt vänster och höger delträd (barnen). När vi har en enstaka nod, representerar vi den som ett binärträd med ett värde och tomma delträd till vänster och höger.

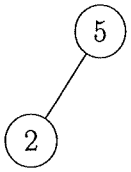
Nedan ser du grafisk representation av ett litet binärträd. Noden 2 har två barn. Vänster delträd är noden 1, och höger delträd är noden 3 (och allt under det). Såväl vänster och höger delträd för noden med värde 1 är det tomma binärträdet.



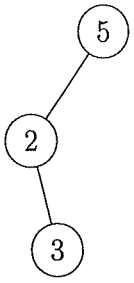
Till 5B: Exempel på insättning i binärt *sök*träd.

Efter insättning av 5:

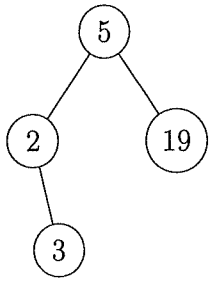
Efter insättning av 2:



Efter insättning av 3:



Efter insättning av 19:



Lämna in dina svar i fil `uppg5.rkt`.

Uppgift 5a) Representation (2p)

Du får själv välja hur den interna representationen av binärträdet ser ut. Följande ska dock implementeras (`bt` står för binärträd):

- Konstruktorn `(make-bt left value right)` som skapar en inre nod. `left`, `right` antas vara delträd.
- Predikatet `(empty-bt? tree)` som testar om ett binärträd är det tomma trädet eller ej.
- Konstanten `null-bt` som representerar det tomma trädet. Även här får du välja hur du representerar det internt.
- Följande getters: `(bt-left node)`, `(bt-right node)`, `(get-value node)` för inre noder.

Se på binärträdet i figuren ovanför, där den översta noden har värde 2. Vi skapar det (och knyter ihop namnet `mytree` med det) med uttrycket

```
(define mytree
  (make-bt
   (make-bt
    null-bt
    1
    null-bt)     ;; vänster delträd
   2             ;; noden har värde 2
   (make-bt
    null-bt
    3
    (make-bt null-bt 4 null-bt)))) ;; höger delträd
```

Uppgift 5b) Binära sökträd, användning (4p)

Vi kan beskriva ett *binärt* ***sökträd*** som ett binärt träd där värdena ordnats på ett särskilt vis. Om värdet av en nod är n, kommer alla värden i vänster delträd att vara mindre än n, och alla värden i höger delträd att vara större. Detta gäller för alla noder.

Exempelträdet på bilden ovan är ett binärt sökträd. Alla värden i vänster delträd från rotnoden är mindre än 2, alla i höger delträd är större. Alla värden i höger delträd från nod med värdet 3 är större.

Din uppgift är att definiera

- `(insert-value bst val)` som tar ett binärt sökträd, ett värde, och returnerar ett nytt binärt sökträd som innehåller värdet på rätt plats.[4]
- `(contains? bst val)` som tar ett binärt sökträd, ett värde, och returnerar `#t` om värdet finns i trädet. Lösningen måste utnyttja att det är ett binärt sökträd.

Använd abstraktionen beskriven i 5a. Vi kan anta att alla värden är tal, som kan jämföras med $<$.

Så här ska det fungera. Eftersom `insert-val` inte ändrar något i trädet (utan skapar ett nytt träd) får vi spara undan värdet under namnet `stree` i varje steg:

```
> (define stree null-bt)
> (set! stree (insert-val stree 5))
> (set! stree (insert-val stree 5))  ;; ingen skillnad
> (set! stree (insert-val stree 2))
> (set! stree (insert-val stree 3))
> (set! stree (insert-val stree 19))
> (get-value stree)
5
> (get-value (bt-left stree))
2
> (get-value (bt-right (bt-left stree)))
3
> (contains? stree 5)
#t
> (contains? stree 511)
#f
```

[4] Om du har läst vidare: notera att vi inte ställer några krav på att trädet ska vara balanserat.

Uppgift 6, Bindningar, omgivningsmodellen (3 poäng)

Lämna in dina svar som kommentarer i fil `uppg6.rkt`.

Vi skriver följande kod:

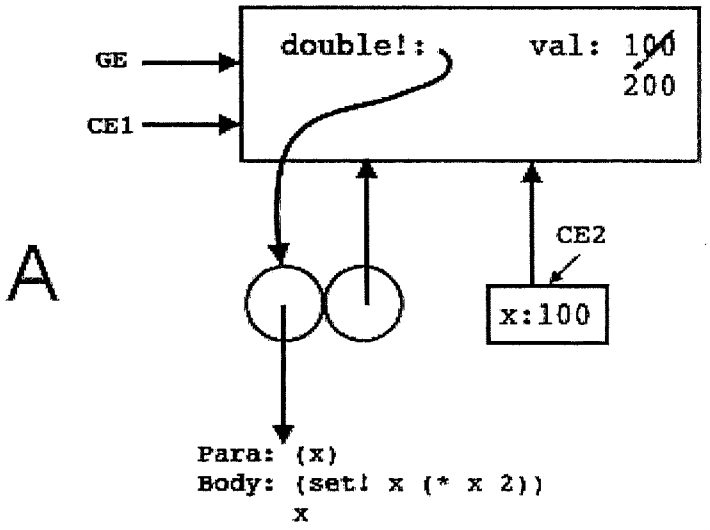
```
(define (double! x)
   (set! x (* x 2))
    x)
(define val 100)
```

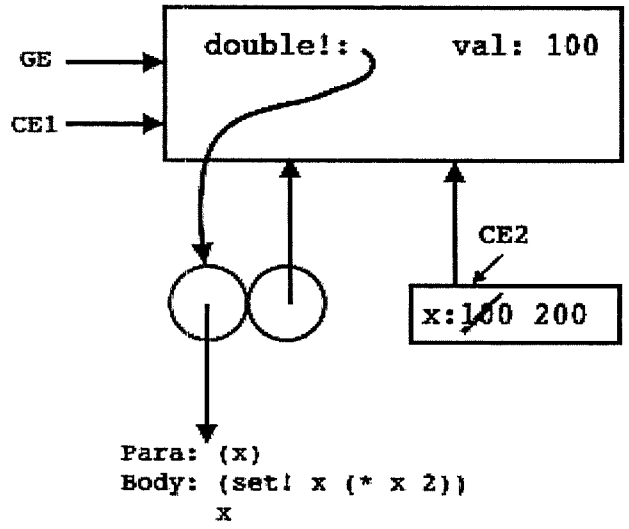
Därefter evaluerar vi uttrycket `(double! val)`.

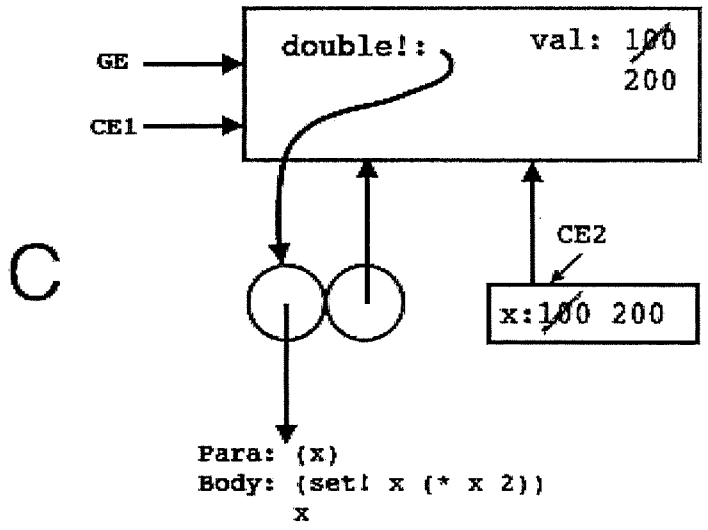
a) Vad händer och varför? Vad för värde returneras? Vad har `val` för värde efteråt? Besvara även om det hade ändrat något om vi skrivit `x` istället för `val` (det vill säga, skrev `(define x 100)`, och anropade med `(double! x)`). (2p)

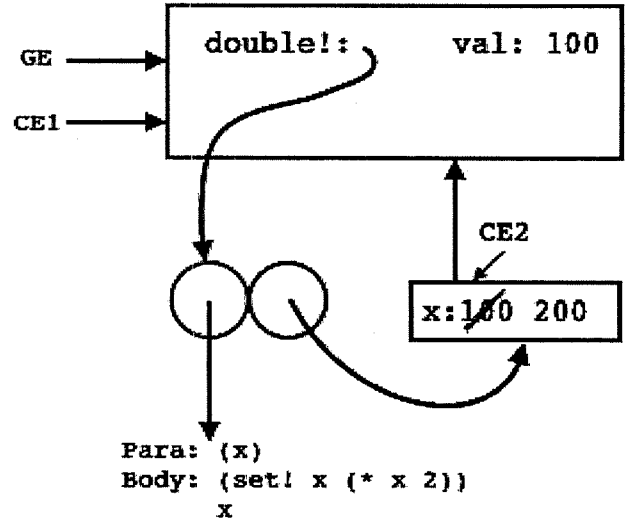
b) Ett av omgivningsdiagrammen nedan illustrerar vad som händer. Vilket? Motivering behövs ej. Korrekt svar ger 1p, felaktigt ger avdrag om 1p[5].

[5]Minsta total poäng på uppgift 6 är 0p.


A
GE
CE1
double!:
val: 100
200
CE2
x:100
Para: (x)
Body: (set! x (* x 2))
x


B
GE
CE1
double!:
val: 100
CE2
x:100 200
Para: (x)
Body: (set! x (* x 2))
x


C
GE
CE1
double!:
val: 100
200
CE2
x:100 200
Para: (x)
Body: (set! x (* x 2))
x


D
GE
CE1
double!:
val: 100
CE2
x:100 200
Para: (x)
Body: (set! x (* x 2))
x