Försättsblad till skriftlig tentamen vid Linköpings universitet

Datum för tentamen	2017-08-26
Sal (1)	SU-salar, IDA(39)
Tid	8-12
Kurskod	TDDC74
Provkod	DAT1
Kursnamn/benämning **Provnamn/benämning**	Programmering - abstraktion och modellering Datortentamen
Institution	IDA
Antal uppgifter som ingår i tentamen	7
Jour/Kursansvarig Ange vem som besöker salen	Jalal Maleki
Telefon under skrivtiden	0706-071963
Besöker salen ca klockan	Är i närheten hela 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 - 2017-08-26

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.

Procedurerna `enumerate`, `map`, `filter`, och `accumulate` som kan ofta förenkla hantering av sekvenser finns i filen **emfa-utility.rkt** på din datortenta i fall du vill använda någon av dem i dina lösningar.

Frågor

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

Att lämna in

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, Problemlösning med rekursion/iteration (4 poäng)

Lämna in din lösning som filen **uppgift-1.rkt**. Svar på b-uppgiften skrivs som kodkommentar i samma fil.

a) (3p) Skriv en procedur `e-stimate` i Scheme som för ett givet $k \geq 0$ beräknar följande summa:

$$\sum_{i=0}^{k} \frac{1}{i!} = \frac{1.0}{0!} + \frac{1.0}{1!} + \frac{1.0}{2!} + \ldots + \frac{1.0}{k!}$$

Där $0! = 1$ och $n! = n \times (n-1) \times \ldots \times 1$.

```
> (e-stimate 0)
1.0
> (e-stimate 1)
2.0
> (e-stimate 2)
2.5
> (e-stimate 10)
2.718281801146385
```

Om man använder heltalet 1 istället för 1.0 i täljaren då får man bråktal som svar som är helt ok.

b) (1p) Är den beräkningsprocess som anrop till `e-stimate` skapar en rekursiv process eller en iterativ process? Motivera kortfattat. Här behöver du inte ta hänsyn till processtypen som fakultetsfunktionen skapar.

Uppgift 2, Högre-ordningens procedur (3 poäng)

Lämna in din lösning som filen **uppgift-2.rkt**.

Skriv en procedur `make-scaler` som tar ett tal k som argument och returnerar en procedur som skall i sin tur ta ett tal x som argument och beräkna $k \times x$.

Så här skall det fungera:

```
> (define doubler (make-scaler 2))

> (doubler 4)
8

> ((make-scaler 3) 7)
21
```

Uppgift 3, Procedur med tillstånd (3 poäng)

Lämna in din lösning som filen **uppgift-3.rkt**.

Definiera en procedur `arg-collector` som tar ett argument. Om argumentet är symbolen `so-far` returneras en lista som innehåller alla tidigare argument, annars läggs argumentet i listan.

Proceduren skall fungera enligt nedan. Som exemplen visar hamnar senaste argumentet först på listan.

```
> (arg-collector 'so-far)
'()
> (arg-collector 10)
10
> (arg-collector '(a))
'(a)
> (arg-collector 'so-far)
'((a) 10)
> (arg-collector 23)
23
> (arg-collector 'so-far)
'(23 (a) 10)
```

Uppgift 4, Dataabstraktion (5 poäng)

Lämna in din lösning som filen **uppgift-4.rkt**.

Vi vill behandla information om vilka filmer olika personer har sett. Därför skapar vi en dataabstraktion som håller ihop information om en person - dess namn (`name`), födelsedag (`birthday`) och en lista på vilka filmer personen har sett (`movies`).

a) (2p) Välj en lämplig representation och definiera följande funktioner för dataabstraktionen.

   1) Konstruktor `(make-person name birthday movies)`
   2) Selektorer `person-name`, `person-birthday`, `person-movies`

   Så här tänker vi använda dataabstraktionen:

```
> (define person-1
    (make-person
       '(Linnea Axberg)
       920311
       '((room) (spartacus) (the revenant)
         (local hero))))

> (person-name person-1)
'(Linnea Axberg)

> (person-birthday person-1)
920311

> (person-movies person-1)
'((room) (spartacus) (the revenant) (local hero))))
```

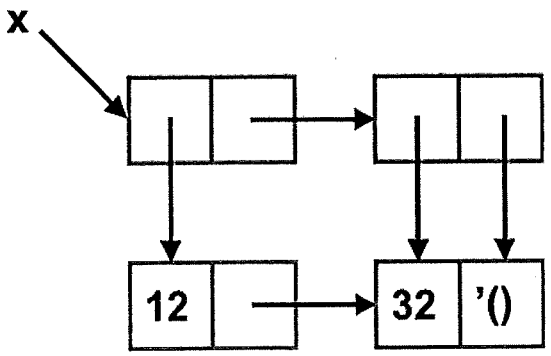
b) (3p) Skriv en funktion `common-movies` som tar två personer som argument och returnerar en lista som innehåller de filmer som båda personerna har sett. Om personerna inte har några filmer gemensamt returneras `'()`.

```
> (define person-2
    (make-person
       '(Anna Persson)
       960104
       '((iceman) (the big short) (jaws)
         (the revenant) (spotlight))))
> (common-movies person-1 person-2)
'((the revenant))
```

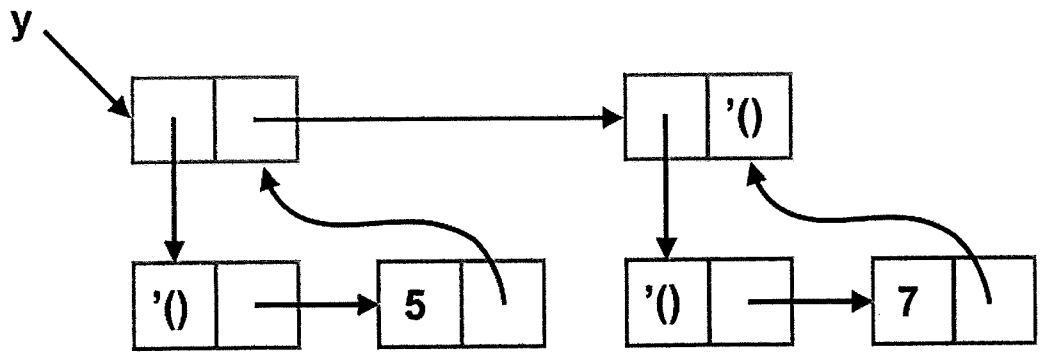
Uppgift 5, Liststrukturer, grafisk representation (3 poäng)

Lämna in din lösning som kodkommentarer i filen **uppgift-5.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.



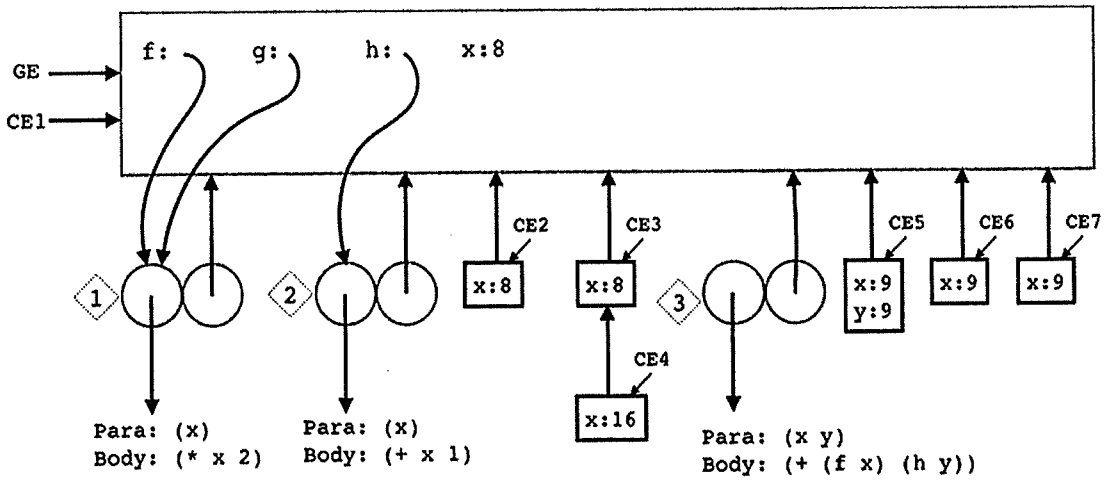
OBS: Pilar till cons-celler pekar alltid på hela paret och inte car- eller cdr-delen.

Uppgift 6, Omgivningsdiagram (3 poäng)

Lämna in din lösning som kodkommentar i filen **uppgift-6.rkt**.

Följande sekvens av uttryck evalueras i den ordningen de förekommer nedan. Vi har ritat ett omgivningsdiagram som innehåller tre fel. Påpeka dessa fel och skriv kort om varför det är fel. Procedurerna har numrerats så att du kan hänvisa till dem lättare. Ramar är också märkta som vanligt (CE1, CE2, etc).

```
(define (f x)
  (* x 2))

(define g f)

(define h (lambda (x) (+ x 1)))

(define x 8)

(f x)

(g (g x))

(let ((x 9)
      (y x))
  (+ (f x) (h y)))
```



Uppgift 7, Bankkonton, något större exempel (5 poäng)

Lämna in din lösning som filen **uppgift-7.rkt**.

Följande program implementerar enkla bankkonton för personer. Varje konto skapas med ett namn (`account-id`) och ett saldo (`balance`). Proceduren `make-account` tar ett konto-id och initialt saldo för ett konto och returnerar en procedur (`dispatch`) som sköter bankärenden för det kontot. För enkelhets skull antar vi att `make-account` alltid får ett unikt konto-id, därför behöver make-account inte testa huruvida ett konto med den identiteten redan finns eller inte.

För att kunna få tag i alla konton vid behov, har vi en global variabel `*account-list*` som innehåller alla konton. Den lista uppdateras genom att `make-account` anropar `add-to-account-list`.

Den programkod som du skall använda finns i filen **bank-accounts.rkt** i din tentakonto. Koden finns på sista sidan av tentan också.

Initialt finns inga konton:

```
> *account-list*
'()
```

Vi skapar fyra konton.

```
> (make-account 'anna 400)
#<procedure:dispatch>

> (make-account 'anders 200)
#<procedure:dispatch>

> (make-account 'lisa 500)
#<procedure:dispatch>

> (make-account 'rumi 600)
#<procedure:dispatch>
```

Nu ser vi att `*account-list*` innehåller fyra konton.

```
> *account-list*
'(#<procedure:dispatch> #<procedure:dispatch> #<procedure:dispatch>
#<procedure:dispatch>)

;find-account can be used to fetch an account using account-ID
> (find-account 'rumi)
#<procedure:dispatch>
```

Din uppgift är följande:

a) (3p) Skriv en procedur `has-more-than` som tar ett belopp som argument och returnerar namnen (ID) på konton som har ett saldo som är strikt högre än beloppet.

```
> (has-more-than 400)
'(rumi lisa)

> (has-more-than 200)
'(rumi lisa anna)
```

b) (2p) Vi vill spara de transaktioner som sker i ett konto. Ett naturligt sätt är att spara transaktionerna i en lokal lista i varje konto. Listan utökas allteftersom vi utför insättningar och uttag. Ett uttag på 20 kronor kan sparas som `'(withdraw 20)` och en insättning med samma belopp som `'(deposit 20)`.

Så här skulle det kunna gå till sedan:

```
> (withdraw 'rumi 20)
580
> (deposit 'rumi 30)
610
> (withdraw 'rumi 50)
560
> ((find-account 'rumi) 'transactions)
'((withdraw 50)
  (deposit 30)
  (withdraw 20))
```

Utöka `make-account`-koden så att den stödjer detta. Transaktionerna skall komma i ordningen senast-först.

```
#lang racket

(provide (all-defined-out))

(define *account-list* '())

(define (add-to-account-list new-account)
  ;Adds the new account to the *account-list*
  (set! *account-list* (cons new-account *account-list*)))

(define (make-account account-id balance)
```

```
  (define (withdraw amount)
    (if (>= balance amount)
        (begin  (set! balance (- balance amount))  balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch message . args)
    (cond ((eq? message 'account-id) account-id)
          ((eq? message 'balance) balance)
          ((eq? message 'withdraw) (withdraw (car args)))
          ((eq? message 'deposit) (deposit (car args)))
          (else (error "Unknown request" message))))
  (add-to-account-list dispatch)
  dispatch)

(define (find-account account-id)
  ;This procedure takes an account ID as argument and finds the
  ;corresponding account in the global variable *account-list*.

  (define (find-help account-list)
    (cond ((null? account-list) (error "Account Not Found!" account-id))
          ((equal? account-id ((car account-list) 'account-id))
           (car account-list))
          (else (find-help (cdr account-list)))))
  (find-help *account-list*))

;Following procedures are not necessary but make it easier
;to work with accounts.

(define (balance account-id)
  ((find-account account-id) 'balance))

(define (withdraw account-id amount)
  ((find-account account-id) 'withdraw amount))

(define (deposit account-id amount)
  ((find-account account-id) 'deposit amount))
```