Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2014-06-11
Sal (2) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	TER1 TERE
Tid	14-18
Kurskod	TDDC74
Provkod	TEN1
Kursnamn/benämning **Provnamn/benämning**	Programmering - abstraktion och modellering Skriftlig tentamen/duggor
Institution	IDA
Antal uppgifter som ingår i tentamen	6
Jour/Kursansvarig Ange vem som besöker salen	Johannes Schmidt
Telefon under skrivtiden	07 25 72 18 00
Besöker salen ca kl.	kl. 15 och 17:
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	Valvritt
Antal exemplar i påsen	

TDDC74 Programmering: Abstraktion och modellering
Tentamen, Onsdag 11 juni 2014, kl 14–18

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 skriva ytterligare hjälpfunktioner som kan vara nödvändiga.

Betygsgradering:

12 – 15,5	betyg 3
16 – 19,5	betyg 4
20 – 24	betyg 5

Lycka till!

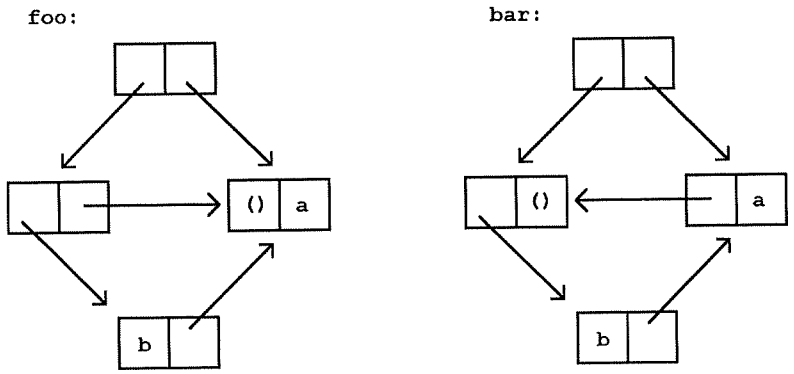
Uppgift 1. Beräkning av uttryck (4 poäng)

Vi ger DrRacket följande uttryck att beräkna (i ordningen som anges).
Vilka värden returneras. Om det blir fel, beskriv varför (vilken typ av fel).

```
> (define a (+ (cons 0 (+ 4 2))) 0)
> a
> (define x '(1 x 3))
> x
> (define b (cadr x))
> b
> (define c (apply (lambda (a b c) (- a c)) x))
> c
> (define f (lambda (n m) (- n (car x))))
> f
> (define h (lambda (a b c) (f b f)))
> h
> (define z (h 1 2 3))
> z
> (define w (f (f h 42) h))
> w
```

Uppgift 2. Box-pointer diagram (4 poäng)

En av strukturerna nedan kan skapas utan muterbara strukturer (alltså enbart med `cons`, `car` och `cdr`). Den andra kräver att man använder `mcons`, `mcar`, `mcdr`, `set-mcar!` och `set-mcdr!`. Skriv kod som skapar strukturerna. Du får bara använda muterbara par när du bygger den av strukturerna som kräver det.



Förtydligande: i såväl `foo` som `bar` pekas det på hela par (inte enbart `car`- eller `cdr`-delen).

Uppgift 3. Rekursiva procedurer (3 poäng)

Ni ska bygga en specialvariant av ett program som "bit-shiftar" en sekvens av bitar (ettor och nollor). Vi representerar en bit med heltalen 0 och 1. Vi representerar en sekvens av bitar som en lista med heltal.

Implementera proceduren `bit-shift`, som tar en bit `new-bit`, och en lista med bitar `bit-list`, och returnerar en *bitshiftad version* av `bit-list` (där `new-bit` skiftats in).

Vi får en bitshiftad version genom att skriva `new-bit` *till vänster* om den gamla sekvensen och sedan ta bort det sista elementet i listan.

Körexempel Så här skall den fungera:

```
> (bit-shift 1 '(1 0 1 1))
(1 1 0 1)
> (bit-shift 0 '(1 0 0 1 1))
(0 1 0 0 1)
> (bit-shift 0 '(1))
(0)
```

Begränsning Ni får enbart använda primitiverna `car`, `cdr`, `cons`, `null?` samt special forms `define` och `if`. Konstanter som `'()` får användas fritt.

Ni kan förutsätta att den givna listan inte är tom och såväl `new-bit` som innehållet i listan är korrekt (alltså enbart innehåller 0:or och 1:or).

Uppgift 4. Högre ordningens procedurer och omgivningsdiagram (4 poäng)

En funktion på formen $f(x) = x^k$, där k är ett positivt heltal, kallas ett monom (engelska: monomial). Ett sådant monoms grad (engelska: degree) är k. De fyra första monomen är x, x^2, x^3, x^4 (av graderna 1, 2, 3 respektive 4). De första tre har egna namn: identitet/identity (x), kvadrat/square (x^2) och kub/cube (x^3).

Uppgift 4a. (2 poäng) Skriv funktionen `make-monomial`, som tar ett heltal `degree` som argument, och returnerar en funktion som representerar monomet av grad `degree`.

Körexempel Så här skall det fungera:

```
(define id (make-monomial 1))
(define square (make-monomial 2))
(define cube (make-monomial 3))
(define degsix (make-monomial 6))

> (id 7)
7
> (square 3)
9
> (cube 3)
27
> (degsix 2)
64
```

Uppgift 4b. (2 poäng) Rita upp omgivningsdiagram som skapas när man *först* evaluerar din definition av `make-monomial`, och *sedan*

```
(define id (make-monomial 1))
(id 7)
```

Markera globala omgivningen (GE) tydligt, och numrera ramarna i den ordning de skapas (E1, E2,...). Du behöver inte skriva ut all kod i procedurkropparna (body), men det ska tydligt framgå vilken kod du hänvisar till.

Uppgift 5. Begreppsfrågor (3 poäng)

Förklara kortfattat (1-3 meningar per delfråga):

1. Den huvudsakliga eller viktigaste skillnaden mellan *funktionell programmering* och *imperativ programmering*
2. Den huvudsakliga eller viktigaste skillnaden mellan *applicative order evaluation* och *normal order evaluation*
3. Vad *substitutionsmodellen* innebär (och när den går respektive inte går att använda).

Uppgift 6. Abstrakt datatyp med listor (6 poäng)

En stack är en abstrakt datatyp som följer "sist in-först ut" (SIFO). Man lägger till element genom att trycka ned alla tidigare element, och lägga det nya överst (*push*). Det nya elementet är då TOS (Top Of Stack). Man kan plocka bort elementet som är TOS med hjälp av *pop*, och får då tillbaka elementet. Ni ska implementera en stack med följande funktionalitet:

1. `push` - Lägg till ett element på stacken.
2. `pop` - Ta bort nuvarande element som är TOS, och returnera det (om möjligt).
3. `peek` - Returnera elementet som är TOS, utan att ta bort det (om möjligt).
4. `contents` - Returnera hela stackens innehåll som en lista.
5. `size` - Returnera stackdjupet (hur många element finns på stacken).
6. `empty?` - Kontrollera om stacken är tom.

Som en underliggande datastruktur kommer vi att använda en enkel lista och vi representerar den tomma stacken med den tomma listan. Färdigställ kodskelettet nedan:

```
(define (make-stack)
  (let
      ((stack-list '()))

    (define (push element) ...)
    (define (pop) ...)
    (define (peek) ...)
    (define (contents) ...)
    (define (size) ...)
    (define (empty?) ...)

    (define (dispatch . message)
      (if (null? message)
          (error "Command needed!")
          (case (car message)
            [(push) (push (cadr message))]
            [(pop) (pop)]
            [(peek) (peek)]
            [(contents) (contents)]
            [(size) (size)]
            [(empty?) (empty?)]
            [else
             (error "Undefined!" (car message))])))
    dispatch))
```

Uppgift 6a. (5 poäng) Implementera de sex procedurerna. Ni får endast använda de grundläggande listoperanderna `car`, `cdr`, `cons` och `null?` samt `set!` och `if` special formerna. Notera att funktionerna `pop` och `peek` kräver en säkerhetskontroll; är stacken tom ska programmet skriva ut ett lämpligt felmeddelande.

Körexempel Så här ska er stack fungera:

```
> (define my-stack (make-stack))
> (my-stack 'empty?)
#t
> (my-stack 'push 'foo)
> (my-stack 'push 'bar)
> (my-stack 'push 'buh)
> (my-stack 'contents)
(buh bar foo)
> (my-stack 'size)
3
> (my-stack 'peek)
buh
> (my-stack 'pop)
buh
> (my-stack 'contents)
(bar foo)
> (my-stack 'pop)
bar
> (my-stack 'pop)
foo
> (my-stack 'peek)
error: Stack is empty!
```

Uppgift 6b. (1 poäng) Förklara vad ett *gränssnitt* (*interface*) eller en *gränssnittsspecifikation* (*interface specification*) är, inom ramen för en diskussion om abstrakta datatyper. Ge också ett exempel.