Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2011-01-12
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	TER2
Tid	14-18
Kurskod	TDDA69
Provkod	TENA
Kursnamn/benämning **Provnamn/benämning**	Data- och programstrukturer Tentamen
Institution	IDA
Antal uppgifter som ingår i tentamen	7
Jour/Kursansvarig Ange vem som besöker salen	Anders Haraldsson
Telefon under skrivtiden	Ankn. 1403 eller mobil 0705-147709
Besöker salen ca kl.	15.00 och 17.00
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	
Antal exemplar i påsen	

Linköpings tekniska högskola
Institutionen för datavetenskap
Anders Haraldsson

Tentamen i

TDDA 69 Data och programstrukturer

Onsdag den 12 januari 2011, kl 14-18

Hjälpmedel Inga.

Poänggränser: Maximalt kan erhållas 58p. För godkänt krävs ca 24p.

Jourhavande lärare: Anders Haraldsson

Lycka till!!

Uppgift 1. Substitutionsmodellen och högre ordningens funktioner (5p)

1a. (2p) Vi definierar följande funktion f:

```
(define f
   (lambda (f)
      (lambda (g) (g f))))
```

Visa genom noggrann utveckling med t.ex. substitutionsmodellen vad värdet blir av

```
((f 5) (lambda (x) (+ x 1)))
```

1b. (3p) Funktionen repeated (som ni definierade i laboration 1) kan definieras som

```
(define (repeated f n)
   (if (= n 1)
      f
      (lambda (x) (f ((repeated f (- n 1)) x)))))
```

Visa genom noggrann utveckling med t.ex. substitutionsmodellen vad värdet blir av

```
((repeated (lambda (x) (expt x 2)) 2) 4)
```

Uppgift 2. Förklara kortfattat följande begrepp (7p)

2a. normal order och applicative order
2b. referential transparency
2c. unifiering
2d. icke-deterministiskt program
2e. "continuation procedure"
2f. garbage collection
2g. dynamisk bindning

Uppgift 3. Omgivningsmodellen och objektorientering (10p)

I bilaga 1 finns eval-%scheme som följer *omgivningsmodellen*. För att förstå hur variabler binds till sina värden illustrerar vi detta med *omgivningsdiagram*.

3a. (2p) Förklara först vad ett omgivningsdiagram består utav. Vad är en *ram* (frame), en *omgivning* (environment), var finns en *variabel* och dess *värde*? Vad är ett *procedurobjekt*? När skapas en ram?

3b. (4p) Nedanstående uttryck evalueras i den ordning de ges till Scheme-interpretatorn. Rita ett diagram baserat på omgivningsmodellen som visar vad som händer. Märk ut de viktigaste strukturerna och beskriv i vilken ordning de skapas och försvinner. Använd diagrammet för att ange uttryckets värde.

```
(define (snoc x y) (lambda (m) (m x y)))
(define (rac z) (z (lambda (p q) p)))
(define x (snoc (quote a) (quote b)))
(define result (rac x))
```

3c. (4p) Här visas ett välkänt kodavsnitt i Scheme, som är ett bankkonto där du kan sätta in och ta ut pengar. Beskriv delarna i koden i relation till olika *begrepp* som du finner inom *objektorienterad programmering*.

```
(define (make-account 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 m)
        (cond ((eq? m 'withdraw) withdraw)
              ((eq? m 'deposit) deposit)
              (else (error "Unknown request" m))))
    dispatch)

(define acc (make-account 200))
((acc 'withdraw) 30) ⇒ 170
((acc 'deposit) 50) ⇒ 220
```

Redogör för egenskaper i den objektorienterade paradigmen som vi kan realisera genom att vi använder omgivningsmodellen. Förklara i ett omgivningsdiagram vad acc är för slags värde.

Uppgift 4. Strömmar (6p)

4a. (4p) Beskriv hur *strömmar* implementeras. Hur skiljer de sig från vanliga listor. Varför kan strömmar inte implementeras som vanliga funktioner? I samband med strömmar finns procedurerna cons-stream, stream-car, stream-cdr, force och delay. Förklara dessa.

4b, (2p) Vi har följande två strömprocedurer definierade:

```
(define (add-streams s1 s2)
   (cons-stream (+ (stream-car s1) (stream-car s2))
                (add-streams (stream-cdr s1) (stream-cdr s2))))

(define (stream-map proc st)
   (cons-stream (proc (stream-car st))
                (stream-map proc (stream-cdr st))))
```

Sedan gör vi

```
(define things (cons-stream 1 (add-streams things things)))
(define s (stream-map (lambda (x) (* 2 x)) things))
(stream-car (stream-cdr (stream-cdr s)))
```

Vilken ström har genererats och vad blir värdet av det sista uttrycket?

Uppgift 5. Evaluatorn och parameteröverföring (10p)

5a. (6p) Beskriv parameteröverföringsmodellerna *call-by-value*, *call-by-name* och *call-by-need*? Diskutera hur dessa modeller implementeras i evaluatorn i bilaga 1.

5b. (4p) Vi vill införa typkontroll i %Scheme och vill möjliggöra att för vissa formella parametrar lägga till en procedur som kontrollerar typen av den aktuella parametern.

```
(%define (foo (%number? x) (%list? y) z)
   some-body)
```

I funktionen foo vill vi testa att första parametern x är ett tal och den andra parametern y är en lista. Den tredje parametern y typetestas ej.

Visa hur du i evaluatorn i bilaga 1 kan implementera denna typkontroll. Om argument ej är av rätt typ får evaluatorn avbryta beräkningen och ge en lämplig utskrift, t.ex. som anger första parameter och argumentvärde som ej uppfyller typkontrollen.

Uppgift 6. Språkutvidgning och makrohantering (8p)

Vi önskar i %Scheme en konstruktion %define-before som utökar en redan existerande funktion genom att först köra funktionskroppen som anges av %define-before och därefter den gamla funktionskroppen.

Exempel:

```
%==> (%define (sum x y) (%+ x y))
ok
%==> (sum 1 2)
3
%==> (%define-before (sum x y) (%display "HEJ "))
ok
%==> (sum 1 2)
HEJ 3
```

6a. (4p) Antag att vi har ett makropaket installerat i %sScheme. Definiera en makro som visar hur konstruktionen kan översättas till "vanlig" %scheme-kod.

6c. (4p) Diskutera hur vi kan implementera ett makropaket. Antag att våra makroprocedurer har form precis som vanliga procedurer och finns lagrade i omgivningen på samma sätt men är taggade som makroprocedurer. Vi kan testa om en procedur är en makroprocedur med

```
(macro-procedure? proc)
```

Visa med kod var du lägger in makro-utvecklingen i evaluatorn för %Scheme i bilaga 1.

Uppgift 7. Språkutvidgning och evaluatorerna (12p)

Vi önskar i den meta-cirkulära evaluatorn %Scheme implementera proceduren repeat, med syntaxen

(repeat test exp_1 exp_2 ... exp_n)

och med evalueringsordningen av argumenten så att vi först beräknar test. Om värdet är sant beräknas satserna exp_i. Detta repeteras tills test är falskt. Som värde returneras done.

Kod för %Scheme med utgångspunkt eval-definitionen finns i bilaga 1.

Vi inför abstraktionsfunktionerna:

```
(define (repeat? exp) (tagged-list? exp 'repeat))
(define (repeat-test exp) (cadr exp))
(define (repeat-body exp) (cddr expr))
```

7a. (2p) Implementera repeat först som en syntaxutvidgning, som översätter repeat till ett if-uttryck och evaluerar if-uttrycket. Vi lägger in i evaluatorn ett nytt fall

```
((repeat? exp) (eval (repeat->if exp) env))
```

Definiera repeat->if.

7b. (4p) Som alternativ kan vi implementera repeat som en special form enligt följande

```
((repeat? exp) (ev-repeat exp env))
```

Definiera ev-repeat.

7c. (6p) Implementera proceduren repeat, enligt modell 7b, i interpretatorn för %Scheme med explicit kontroll enligt bilaga 2. Eftersom inte bilagorna innehåller fullständig kod så kan du införa egna primitiver. Förklara bara vad de gör.

BILAGA 1

```
;; Core of the evaluator

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error 'eval "Unknown expression type ~s" exp))))

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error 'apply "Unknown procedure type ~s"
                procedure))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

BILAGA 2

```
evaldispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
.....
  (test (op if?) (reg exp))
  (branch (label ev-if))
.....
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))

ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg
env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  .....
  (goto (reg continue))

ev-application
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
ev-appl-did-operator
  (restore unev)
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val))
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  (save env)
  (save unev)
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
ev-appl-accumulate-arg
  (restore unev)
  (restore env)
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (assign unev (op rest-operands) (reg unev))
  (goto (label ev-appl-operand-loop))
ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore proc)
  (goto (label apply-dispatch))
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))

primitive-apply
  .....

compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
        (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))

ev-begin
  (assign unev (op begin-actions) (reg exp))
  (save continue)
  (goto (label ev-sequence))

ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))

ev-if
  (save exp)
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch))
ev-if-decide
  (restore continue)
  (restore env)
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-if-consequent))
ev-if-alternative
  (assign exp (op if-alternative) (reg exp))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg exp))
  (goto (label eval-dispatch))
  .....
```