Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2012-01-12
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	U1
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 070-514 77 09
Besöker salen ca kl.	ca kl. 15
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	

Linköpings tekniska högskola
Institutionen för datavetenskap
Anders Haraldsson

Tentamen i

TDDA 69 Data och programstrukturer

Torsdag den 12 januari 2012, kl 14-18

Hjälpmedel Inga.

Poänggränser: Maximalt kan erhållas 64p. För godkänt krävs ca 25p.

Examinator: Anders Haraldsson

Lycka till!!

Uppgift 1. Beräkningsordning och parameteröverföring (10p)

1a. (2p) Beräkning av uttryck kan göras med *applicative order* och *normal order*. Vad innebär de båda beräkningssätten? Ger de alltid samma resultat? Om inte ge ett exempel där resultatet skiljer sig åt.

Hur är parameteröverföringsmodellerna *call-by-value* och *call-by-name* relaterade till dessa beräkningsordningar?

1b. (1p) Vad skiljer parameteröverföringsmodellerna *call-by-name* och *call-by-need*?

1c. (3p) Vi har följande uttryck:

```
(define n 10)

(define (g a b c)
  (if (< a 10) (+ a b) (+ a c c)))

(define (f n) (/ 100 n))

(g (+ n 5) (f 0) (begin (set! n (+ n 1)) n))
```

Vad får vi för resultat av detta uttryck om det språk, som beräknar detta, har följande parameteröverföring:

I) *call-by-value*
II) *call-by-name*
III) *call-by-need*

Förklara även varför du fick de resultat du angett.

1d. (4p) I Ada och C har man möjlighet att överföra parametrar med *call-by-reference*. I Ada har vi **out**-deklarerade variabler. Vada innebär *call-by-reference*?

Där kan man definiera en funktion `switch` som byter värdet på två parametrar.

```
(define (switch x y)
  (let ((temp y))
    (set! y x)
    (set! x temp)))
```

Vi kan då byta värden på variablerna `a` och `b`.

```
(define a 10)
(define b 20)
(switch a b)

a = 20
b = 10
```

Kan vi med någon av de olika modellerna för parameteröverföring definiera denna funktion direkt eller måste vi gå in i eval-modellen (bilaga 1) och göra ändringar där? Beskriv hur du skulle göra för att få *call-by-reference*.

Uppgift 2. Omgivningsmodellen och statisk & dynamisk bindning (14p)

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

2a. (1p) En beräkningsmodell av ett språk kan följa *substitutionsmodellen*. Vad är det och varför inför vi *omgivningsmodellen*? Vad är det för konstruktioner som vi vill kunna hantera genom omgivningsmodellen?

2b. (2p) I omgivningsmodellen skapas *procedurobjekt*. Vad består det utav och vad vill man åstadkomma med objektet? Vilket uttryck i Scheme/%Scheme skapar ett sådant objekt. Ge ett exempel på ett sådant uttryck.

2c. (4p) Nedanstående uttryck evalueras i den ordning de ges till Scheme-interpretatorn. Vad kommer värdet av det sista uttrycket att vara. 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 (f x) (g (lambda (y) (+ x y)) 6))

(define (g h x) (h (+ n x))))

(f 1)
```

2d. (1p) Vilket värde skulle vi få om en Lisp-interpretator i uppgift 2b istället använde *dynamisk bindning*?

2e. (6p) Vi definierar en funktion monitor för att hålla reda på vilka argument vi har anropat en funktion med och totalt antalet gånger funktionen har anropats och ger nedanstående uttryck till Scheme.

```
(define (monitor proc)
  (let ((args '())
        (calls 0))
    (lambda (arg)
      (set! args (cons arg args))
      (set! calls (+ 1 calls))
      (cond ((eq? arg 'calls) calls)
            ((eq? arg 'args) args)
            (else (proc arg))))))

(define m-mult (monitor (lambda (args) (apply * args))))

(m-mult '(2 3 4))
(m-mult '(5 6))
(m-mult 'calls)
```

Rita upp omgivningsdiagram och förklara vad som händer när dessa uttryck beräknas av Scheme. Det skall framgå i vilken ordning ramar skapas och du skall kunna ange vilka ramar som ej längre behövs när uttrycken är beräknade (som en garbage collector skull kunna ta bort).

Uppgift 3. Strömmar (6p)

Strömmar är en kraftfull struktur med intressanta egenskaper. I samband med strömmar definierades följande 5 operationer cons-stream, stream-car, stream-cdr, force och delay.

3a. (2p) Beskriv vad *strömmar* är och vad som skiljer det från vanliga listor. Ge exempel på några typer av problem som är lämpliga att modellera med strömmar. Diskutera för- och nackdelar med att använda strömmar resp. vanliga listor.

3b. (2p) Visa hur man implementerade de fem strömprimitiverna i Scheme (t ex som i laborationen) , där eval-modellen som beräknar parametrar är *call-by-value* (såsom i bilaga 1) och har *makroprocedurer / extended syntax.*

3c. (2p) Nedanstående uttryck ges till en evaluator, som kan hantera strömmar. Förklara vad som händer med respektive uttryck. Vilka värden genererar då strömmen mystary?

```
(define ones (cons-stream 1 ones))

(define (integers-from n)
   (cons-stream n (integers-from (+ n 1))))

(define integers (integers-from 1))

(define (multiply-stream s1 s2)
   (cons-stream (* (stream-car s1) (stream-car s2))
                (multiply-stream (stream-cdr s1) (stream-cdr s2))))

(define mystery
   (cons-stream 1 (multiply-stream integers mystery)))
```

Uppgift 4. Makrohantering (6p)

Vad innebär makrohantering? Vad är det för egenskaper i Lisp-språken som gör att det är enkelt att implementera makrohantering? Vad kan man vinna på att ha makrohantering i ett programspråk. Ge exempel. Beskriv problem som kan uppstå när man använder makroprocedurer.

Ge exempel på olika modeller på hur man kan implementera makrohantering. När kan makroutvecklingen utföras?

Uppgift 5. Logikprogrammering och icke-deterministisk programmering (10p)

5a. (2p) I logikprogrammering används *mönstermatchning* och *unifiering*. Förklara vad det är och vad det används till i logikprogrammeringen.

5b. (2p) Implementeringen av QLOG gjorde med hjälp av strömmar. Förklara vad strömmarna innehåller och hur strömmarna skapas och förändras.

5c. (6p) I den icke-deterministiska evaluatorn ambeval inför man möjligheten att resultatet av beräkning av ett uttryck "misslyckas", vilket resulterar i att vi skall göra "back-tracking" till en tidigare valpunkt.

Implementeringen av den icke-deterministiska Scheme-interpretatorn ambeval görs med hjälp av *continuations*. Toppfunktionen ambeval definierades enligt följande:

```
(define (ambeval exp env succeed fail)
   ((analyze exp) env succeed fail))
```

Förklara vad continuations innebär. Vad gör analyze och förklara vad argumenten till ambeval är.

Nedanstående kodavsnitt implementerar en förenklad variant av begin, kallad begin2, som bara tar två argument, beräknar dessa och ger som resultat värdet av det andra argumentet.

```
(define (analyze-begin2 body)
   (let ((a (analyze (car body)))
         (b (analyze (cadr body))))
     (lambda (env succeed fail)
        (a env
           (lambda (a-value fail2)
              (b env succeed fail2))
           fail))))
```

Förklara koden och ge några exempel som illustrerar vad som händer.

Uppgift 6. Rekursion (8p)

6a. (4p) Vi har lärt oss att Scheme hanterar *svansrekursion* i interpretatorn och att andra Lisp-språk (och även andra språk) hanterar svansrekursion vid kompilering. Vad menas med svansrekursion? Ge exempel på en procedur som är svansrekursivt och en som inte är det. Vad kan man vinna genom att hantera svansrekursion på ett eget sätt?

6b. (4p) I den explicita kontrollevaluatorn fanns kod för de båda varianterna. Kan du, med hjälp av ett exempel, förklara med nedanstående kod vad som händer i de båda fallen.

Vad skall ?? i (goto ??) ersättas med i de båda fallen.

```
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 ??)
```

```
ev-sequence
 (test (op no-more-exps?) (reg unev))
 (branch (label ev-sequence-end))
 (assign exp (op first-exp) (reg unev))
 (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-end
 (restore continue)
 (goto ??)
```

Uppgift 7. Evaluatorerna (10p)

7a. (4p) Vi önskar i %Scheme implementera en "special form" för en repetitionsstruktur until, enligt följande struktur:

```
(until end-condition
       expr
       expr
       ...
       expr)
```

Kroppen, dvs sekvensen av uttrycken expr, beräknas först. Sedan beräknas end-condition. Om detta uttryck blir sant så avslutas repetitionen, annars forsätter vi med att beräkna kroppen igen etc. Värdet av until-uttrycket är värdet av det sist beräknade expr-uttrycket. Exempel i %Scheme.

```
(until (> i limit)
  (process i)
  (%set! i (%+ i 1))
  i)
```

utför en vanlig repetition tills i är större än limit. Värdet som returneras är sista värdet av i. Definiera until i %Scheme med utgångspunkt eval-definitionen i bilaga 1 (4p).

7b. (6p) Implementera until 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

```
eval-dispatch
  (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))
.....
```