Försättsblad till skriftlig tentamen vid Linköpings Universitet

(fylls i av ansvarig)

Datum för tentamen	*2010-01-14*
Sal	*-*
Tid	*14-18*
Kurskod	*TDDA69*
Provkod	*TENA*
Kursnamn/benämning	*Data- och programstrukturer*
Institution	*IDA*
Antal uppgifter som ingår i tentamen	*8*
Antal sidor på tentamen (inkl. försättsbladet)	*8*
Jour/Kursansvarig	*Peter Dalenius*
Telefon under skrivtid	*281427 alt 0706601564*
Besöker salen ca kl.	*16.00*
Kursadministratör (namn + tfnnr + mailadress)	*Anna Grabska Eklund Ankn. 23 62, annek@ida.liu.se*
Tillåtna hjälpmedel	
Övrigt (exempel när resultat kan ses på webben, betygsgränser, visning, övriga salar tentan går i m.m.)	
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

Torsdag den 14 januari 2010, kl 14-18

Hjälpmedel Inga.

Poänggränser: Maximalt kan erhållas 60p. För godkänt krävs ca 24p.

Lycka till!!

Uppgift 1. Förklara kortfattat följande begrepp (4p)

1a. referential transparency
1b. constraint propagation
1c. unifiering
1d. garbage collection

Uppgift 2. Beräkningsmodeller (10p)

2a. (2p) Beräkningsordningen 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.

2b. (2p) Vad skiljer parameteröveringsmodellerna *call-by-need* och *call-by-name*? Vilken av beräkningsordningarna i uppgift 2a motsvaras av dessa parameteröverföringsmodeller? Ge exempel där parameteröverföringsmodellerna ger olika resultat.

2c. (6p) Varför kan inte *strömmar* (streams) direkt implementeras i ett språk där man utför parameteröverföringen med *call-by-value*? Hur löstes det i Scheme för att göra labbarna? I samband med strömmar fanns procedurerna cons-stream, stream-car, stream-cdr, force och delay.

Beskriv olika sätt att implementera strömmar (exempelvis hur du gjort på laborationerna) och illustrera, om det är möjligt, med hur du skulle göra det i eval-modellen i bilaga 1.

Uppgift 3. Omgivningsmodellen och bindningsmodeller (12p)

I nedanstående uppgifter vill vi att du ritar omgivningsdiagram. Du skall ange i vilken ordning ramar, bindningar och skapandet av procedurobjekt utförs. Du skall även ange vilka ramar och objekt som efter beräkningen ej längre är åtkomliga och därmed kunna tas bort.

3a. (2p) Börja med att förklara vad *statisk bindning* innebär.

3b. (2p) Rita omgivningsdiagram för

```
(define (make-x y)
   (lambda (a) (+ a y))
   (* 2 y))

(define x (make-x 5))
```

3c. (2p) Rita omgivningsdiagram för

```
(define (make-x y)
   (define p (lambda (a) (+ a y)))
   (display (* 2 y))
   p)

(define x (make-x 5))
(define y (x 10))
```

3d. (4p) Rita omgivningsdiagram för

```
(define (listing a)
   (let ((b (lambda (c) (c))))
      (list a (a) b (b a))))

(define s (listing (lambda () 8)))
```

3e. (2p) Om vi istället för statisk bindning har *dynamisk bindning*. Förklara vad detta innebär. Vad skiljer när du ritar ett omgivningsdiagram?

Vad får vi för resultat om Scheme hade dynamisk bindning i nedanstående exempel.[1]

```
(define (f x)
   (define (g y) (+ y x))
   (define (h x) (+ x (g x)))
   (display (g 3))
   (display (h 3)))

(f 2) => ?
```

Uppgift 4. Objektorientering (4p)

Visa hur man implementerar en objektorienterad modell i Scheme med hjälp av procedurobjekt. Ge ett exempel där du t ex implementerar en *klass* person, med några olika *attrubut* (t ex namn, ålder och barn) och *metoder* (t ex få reda på åldern, öka åldern, lägga till ett barn). Visa sedan hur man kan skapa ett *objekt* av denna klass och använda dess metoder.

Uppgift 5. 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 6. Icke-determinism och continuations (6p)

Den *icke-deterministiska* Scheme implementeras med hjälp av *continuations*. Förklara följande:

6a. (2p) Vad menas med den *icke-deterministiska* programmeringsparadigmen? Ge exempel på problem som den är lämplig för.

6b. (4p) 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.

---

1. Detta skulle vara möjligt att få i Common Lisp med special-deklarationen.

Uppgift 7. Rekursion (8p)

7a. (2p) 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?

7b. (6p) 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 8. Evaluatorerna (10p)

8a. (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).

8b. (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

```
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 ??) ; uppgift 6c

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))
.....
```