Försättsblad till skriftlig tentamen vid Linköpings Universitet

(fylls i av ansvarig)

Datum för tentamen	*2010-08-19*
Sal	*-*
Tid	*14-18*
Kurskod	*TDDA69*
Provkod	*TENA*
Kursnamn/benämning	*Data- och programstrukturer*
Institution	*IDA*
Antal uppgifter som ingår i tentamen	*6*
Antal sidor på tentamen (inkl. försättsbladet)	*7*
Jour/Kursansvarig	*Anders Haraldsson*
Telefon under skrivtid	*281427*
Besöker salen ca kl.	*15.30*
Kursadministratör (namn + tfnnr + mailadress)	*Anna Grabska Eklund* *Ankn. 23 62, annek@ida.liu.se*
Tillåtna hjälpmedel	*Inga*
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 19 augusti 2010, kl 14-18

Hjälpmedel Inga.

Poänggränser: Maximalt kan erhållas 58p. För godkänt krävs ca 25p.

Jourhavande lärare: Anders Haraldsson

Lycka till!!

Uppgift 1. Substitutions- och omgivningsmodell (12p)

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

1a. (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?

1b. (3p) Antag att vi har ett uttryck som ges till eval (i bilaga 1) i en omgivning env. Du skall förklara hur omgivningsdiagrammet används (om det behöver användas) för följande olika typer av Scheme-uttryck:

I) ett självevaluerande uttryck, t ex
```
(eval '10 env)
```

II) ett variabelnamn, t ex
```
(eval 'a env)
```

III) definition av en variabel med define, t ex
```
(eval '(define a 20) env)
```

IV) tilldelning av en variabel med set!, t ex
```
(eval '(set! a 10) env)
```

V) beräkning av ett lambda-uttryck, t ex
```
(eval '(lambda (x) (+ x y)) env)
```

VI) beräkning av en applikation, ett funktionsuttryck, t ex
```
(eval '(foo 3 5) env)
(eval '((lambda (x y) (+ x y 10)) 2 5) env)
```

1c. (8p) Antag att funktionerna f och g definieras enligt nedan. Vad kommer värdet för uttrycket (f 1) bli om vi använder

a) statisk bindning
b) dynamisk bindning

Rita i vardera fallet ett omgivningsdiagram baserat på omgivningsmodellen. Märk ut de viktiga strukturerna och beskriv tydligt varför, och i vilken ordning, de skapas och försvinner. Använd diagrammen för att ange hur värdena har beräknats i de två fallen.

```
(define (g h n)
   (let ((x 5))
      (h (+ n x))))

(define (f x)
   (g (lambda (y) (- (+ x y) 2)) 6))
```

Uppgift 2. Parameteröverföring (12p)

2a. (4p) Beskriv parameteröverföringsmodellerna *call-by-value*, *call-by-name* och *call-by-need*? Diskutera dessa modeller hur de skulle kunna implementeras i evaluatorn i bilaga 1. Fullständig kod behöver inte ges, utan visa var i koden man måste gör ändringarna.

2b. (3p) Om man har ett språk med i grunden parameteröverföringsmodell *call-by-value*, varför kan man inte själv implementera alla primitiverna för *strömmar* (streams) direkt i ett sådant språk?

Om man i ett sådant språk har tillgång till *makroutveckling* (enligt den modell som vi använd i denna kurs), hur kan man då själv implementera alla primitiverna för strömmar.

2c. (5p) 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 z typtestas ej.

```
(foo 10 (%quote (q w e)) 20)    är ok
(foo (%quote a) 10 20)          har typfel
```

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 3. Olika beräkningsmodeller (14p)

3a. (3p) I logikprogrammering används *mönstermatchning* och *unifiering*. Vad är det? Ge exempel på lyckade och inte lyckade unifieringar. När har man behov i logikprogrammering (t ex QLOG) att utföra mänstermatchning resp. unifiering?

3b. (3p) Definiera i Prolog eller QLOG ett predikat, prefix, som avgör om en lista är ett prefix till en annan lista.

(prefix (1 2) (1 2 3))	är sant
(prefix (2 3) (1 2 3))	är falskt
(prefix (2 1) (1 2 3))	är falskt

3c. (8p) I den icke-deterministiska Scheme (amb-evaluatorn) inför man möjligheten att resultatet av en beräkning "misslyckas", vilket resulterar i att vi skall göra "back-tracking" till en tidigare valpunkt. Implementeringen av den icke-deterministiska Scheme-interpretatorn görs med hjälp av *continuations*.

Man inför en ny operation amb. Vad gör den?

Förklara sedan vad *continuation* betyder. Vad skiljer det sig från vanlig beräkning av funktionsanrop?

Hur kan man "backa" i koden? Vad är en *failure-continuation* och vad består den utav. Förklara dess roll hur man kan hitta nästa alternativ i en tidigare valpunkt.

Vad händer med redan gjorda sidoeffekter när man "backar" tillbaka. Kan dessa bli ogjorda?

Nedanstående kodavsnitt implementerar en förenklad variant av begin, kallad seq2 som bara tar två argument, beräknar dessa och ger som resultat värdet av det andra argumentet.

```
(define (analyze-seq2 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 4. Variabelhantering (4p)

Vid implementeringen lagrade vi både namn och värde i ramen och gjorde variabeluppslagningen genom sekventiell sökningen genom ramarna. Har vi statisk bindning kan vi optimera detta. Diskutera denna optimeringsstrategi och hur den kan realiseras i vår eval-modell.

Uppgift 5. Hantering av rekursion (6p)

5a. (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?

5b. (4p) Förklara hur Scheme gör för att hantera svansrekursionen. Kan du visa på ställen i den explicita kontrollevaluatorn i bilaga 2 hur man kan se detta.

Uppgift 6. Evaluatorerna (10p)

Vi önskar implementera en special form case, med följande utseende:

(case (variabel uttryck)
  ($värde_1$ $uttryck_1$)
  ($värde_2$ $uttryck_2$)
  ...
  ($värde_k$ $uttryck_k$))

Ett case-uttryck evalueras så att först evalueras uttryck, vars värde binds till variabel. Sedan jämföres detta värde med $värde_i$, ett i taget. För första $värde_i$ som är lika beräknas $uttryck_i$. I detta uttryck kan variabel användas. Efter beräkningen skall variabel ej längre vara åtkomlig. Finns ingen värde returneras nil.

(case (i (+ 3 5))
  (3 (+ i 5))
  (8 (+ i 10))
  (11 (+ i 20))) = 18 , *dvs andra fallet. Variabeln* i *binds till 8, vi får 8+10.*

6a. (4p) Utvidga %scheme med case i den grundläggande interpretatorn i bilaga 1.

6b. (6p) Implementera case 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.

```
;; 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))
 .....
```