Försättsblad till skriftlig tentamen vid Linköpings Universitet

(fylls i av ansvarig)

Datum för tentamen	*2009-08-25*
Sal	*T1*
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)	*9*
Jour/Kursansvarig	*Anders Haraldsson*
Telefon under skrivtid	*281403, 0705147709 (mobil)*
Besöker salen ca kl.	*Kl 15 och 17*
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

Tisdag den 25 augusti 2009, kl 14-18

Hjälpmedel Inga.

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

Jourhavande lärare och examinator: Anders Haraldsson, tel 28 14 03 (mobil: 0705 147709)

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. (1p) Förklara först vad ett omgivningsdiagram består utav. Vad är en *ram* (frame), en *omgivning* (environment), var finns *variablerna* och dess *värden*?

1c. (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)
```

1d. (1p) I Scheme har vi let och let* för att introducera lokala variabler enligt följande syntax:

(let/let* ($((var_1\ expr_1)\ (var_2\ expr_2)\ \ldots\ (var_n\ expr_n))$ body)

Vad är det för skillnad på let och let*. Dessa kan transformeras till lambda-uttryck. Visa hur.

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

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

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

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

1f. (4p) Visa i ett omgivningsdiagram vad som skapas då uttrycken i uppgift 1e utförs. Man skall kunna följa i vilken ordning som de olika delarna skapas och i vilken omgivning de olika uttrycken beräknas i och vilka delar som ej längre behövs efter det att uttrycken har beräknats.

Uppgift 2. Objektorientering (4p)

2a. (2p) Här visas ett välkänt kodavsnitt i Scheme, som är ett bankkonto där du kan sätta in och ta ut pengar och är en modell för *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 100))

((acc 'withdraw) 40)
⇒ 60

((acc 'deposit) 30)
⇒ 90
```

Beskriv delarna i koden i relation till följande *begrepp* som du finner inom *objektorienterad programmering*: *klass*, *instans*, *attribut*, *metod* och *meddelandesändning*.

2b. (1p) Vi vill byta syntax till en mer funktionell syntax och skriva på följande sätt istället:

```
(define acc (make-account 100))

(withdraw acc 40)
⇒ 60

(deposit acc 30)
⇒ 90
```

Definiera funktionen withdraw som tar ett konto och ett belopp att ta ut resp. sätta in på kontot.

2c. (1p) Komplettera med en metod så att du kan få reda på kontoställningen.

Uppgift 3. Parameteröverföring (12p)

3a. (6p) 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.

3b. (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.

3c. (3p) 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.

`(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 4. Olika beräkningsmodeller (12p)

4a. (2p) I logikprogrammering används *unifiering*. Vad är det? Ge exempel på lyckade och inte lyckade unifieringar. När har man behov i logikprogrammering (t ex QLOG) att utföra unifiering?

4b. (3p) Definiera i Prolog eller QLOG ett predikat, subset, som avgör om en lista är en delmängd av en annan lista.

```
(subset (2 1) (1 2 3))      är sant
(subset (1 1 2) (1 2 3))    är sant
(subset (2 4) (1 2 3))      är falskt
```

4c. (2p) Nedan skall fibs definieras som en ström av alla fibonacci-talen (de är 0, 1, 1, 2, 3, 5, 8, 13, 21 etc). Vi skall definiera fibs på ett sådant sätt så att vi rekursivt återanvänder fibs. Funktionen add-streams kan vara lämplig att använda som tar två strömmar och adderar motsvarande element som genererar en ny ström. Kan du från ditt exempel sedan förklara hur fibonacci-strömmen skapas, i vilken ordning sker de olika operationerna.

```
(define fibs (cons-stream 0 (cons-stream 1 ... kod ...))

(define (add-streams s1 s2)
   (cond ((stream-null? s1) s2)
         ((stream-null? s2) s1)
         (else (cons-stream (+ (stream-car s1) (stream-car s2))
                            (add-streams (stream-cdr s1) (stream-cdr s2))))))
```

4d. (5p) 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 gjordea 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 5. Hantering av rekursion (8p)

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?

5a. (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.

5c. (2p) Vi definierar en funktion loop som var millionte varv skriver ut ett värde.

```
(define (loop n)
   (cond ((= n 0) (begin (display 'ok) (newline)))
         ((= (remainder n 1000000) 0)(begin (display n) (newline)
                                            (loop (- n 1))))
         (else (loop (- n 1)))))
```

Vi gör (loop 50000000), dvs loop anropas 50 miljoner gånger!

Är det möjligt att beräkna detta? Vad ligger i så fall på stacken i det svansrekursiva resp det icke svansrekursiva fallet. Vi har ändå skapat 50 miljoner ramar och ändå finns minne kvar. Hur är detta möjligt?

Uppgift 6. Evaluatorerna (10p)

6a. (4p) Vi önskar implementera proceduren and, med evalueringsordningen av argumenten så att vi endast beräknar argumenten så länge dess värde är sanna. Efter första argument som beräknas till falskt avslutas sålunda beräkningen av argumenten. Implementera proceduren and i %Scheme med utgångspunkt eval-definitionen i bilaga 1.

6b. (6p) Implementera proceduren and 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))
 .....
```