



Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2010-06-01
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och <u>ringa in</u> vilken sal som avses	U1
Tid	8-12
Kurskod	TDDA69
Provkod	TENA
Kursnamn/benämning Provnamn/benämning	Data- och programstrukturer Tentamen
Institution	IDA
Antal uppgifter som ingår i tentamen	6
Jour/Kursansvarig Ange vem som besöker salen	Anders Haraldsson
Telefon under skrivtiden	ankn. 1403, mobil 070-5147709
Besöker salen ca kl.	?
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, ankn. 2362, annek@ida.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

Tisdag den 1 juni 2010, kl 8-12

Hjälpmedel: Inga.

Poänggränser: Maximalt kan erhållas 60p. För godkänt krävs ca 25p.
Det finns möjlighet till extra poäng. Se efter uppgift 6.

Jourhavande lärare och examinator: Anders Haraldsson, tel 28 14 03, tel 070 5147709

Lycka till!!

Anders Haraldsson
Peter Dalenius

Uppgift 1. Beräkningsmodeller (12p)

- 1a.** (2p) Vi införde tidigt i kursen två beräkningsmodeller, *substitutionsmodellen* och *omgivningsmodellen*. Vad skulle de illustrera?
- 1b.** (1p) Omgivningsmodellen har begrepp som *bindning*, *ram* och *omgivning*. Definiera dessa.
- 1c.** (2p) Vad händer i omgivningsmodellen då vi utför en `define`, `set!`, beräkning av lambda-uttryck och funktionsapplikation/funktionsanrop?
- 1d.** (7p) Visa med hjälp av omgivningsmodellen vad som händer då följande uttryck ges till Scheme för beräkning. Visa i vilken ordning de olika objekten och ramarna skapas och vad som kan tas bort vid en *garbage collection*. Vilket värde erhålls av de två sista uttrycken? Rita omgivningsdiagrammet.

```
(define y 10)

(define (f x)
  (define y 20)
  (g (+ x y)))

(define g
  (let ((x y))
    (lambda (z)
      (set! x(+ y z))
      (set! y (+ x 5))
      (+ x y))))

(f 1)
(f 1)
```

Uppgift 2. Parameteröverföringsmodeller och statisk & dynamisk bindning (10p)

I denna uppgift kan ni tänka er att ni använder %Scheme, dvs ni får då lägga in % före alla Scheme-operatorer.

2a. (4p) Vad innebär *dynamisk bindning*? Visa vilka ändringar du måste göra i eval-modellen i bilaga 1 för att få dynamisk bindning.

Vi har följande uttryck:

```
(define (f x)
  (define (g y) (+ x y))
  (define (h x) (+ x (g 1)))
  (display (g 2)) (newline)
  (display (h 2)) (newline))
```

(f 5)

Vad skrivs ut om vi

- a) har *statiskt* bindning/räckvidd och *call-by-value*
- b) har *dynamisk* bindning/räckvidd och *call-by-value*

2b. (6p) Vad menas med *call-by-name* och *call-by-need*?

Vi har följande uttryck:

```
(define q 10)

(define (f x y z)
  (if (> x 0)
      (+ y x y)
      (+ y z)))

(define (loop) (loop)) ; ger oändlig loop

(f 5 (loop) (begin (set! q (+ q 1)) q))
```

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

- a) Statisk bindning/räckvidd och *call-by-name*
- b) Statisk bindning/räckvidd och *call-by-value*
- c) Statisk bindning/räckvidd och *call-by-need*

Förklara även varför du fick de resultat du angett.

Uppgift 3. Språkutvidgning och strömmar (12p)

Under laborationerna har ni implementerat *strömmar* på olika sätt, beroende på egenskaper i implementeringsspråket (i laboration 2 Scheme, i övriga laborationer %Scheme).

Strömmar hade följande primitiva operationer: `delay`, `force`, `cons-stream`, `stream-car`, `stream-cdr`, `the-empty-stream` och `stream-null?`.

3a. (1p) Vad är strömmar? Vilka egenskaper har en ström? Varför kan inte vanliga listor användas och varför kan det inte implementeras i ett språk med parameteröverföring *call-by-value*?

3b. (1p) Hur implementerades strömmar i laboration 2, då ni utvidgade vanliga Scheme-språket med detta?

3c. (2p) I laboration 3 (uppgift 4) utvidgades %Scheme med strömmar genom att vissa primitiver implementeras som *special form*. Vad innebar detta?

3c. (2p) I laboration 3 (uppgift 5) utvidgades %Scheme med makrohantering. Sedan implementerade ni strömmar (uppgift 8) genom att vissa primitiver implementeras som *makron*. Vad innebar detta?

3d. (2p) I laboration 4 implementerades %Scheme med *lat evaluering*. Då borde det vara möjligt att definiera

```
(%define (%cons-stream a b) (%cons a b))
```

eller alternativt att %cons direkt kan användas för strömmar. Varför fungerade det inte? Vad måste man göra i stället?

3e. (2p) I laboration 4 (uppgift 2) införde vi en evaluator med de tre olika parameteröverföringsmodellerna, *call-by-value*, *call-by-name* och *call-by-need*, där man för varje parameter i en funktionsdefinition kan ange hur den skall hanteras. I parameterlistan använde man nyckelordet %lazy för *call-by-name* och %lazy-memo för *call-by-need*. Hur kan vi nu implementera strömmar?

3f. (2p) Sammanfatta nu mer generellt olika möjligheter att utvidga ett språk med konstruktioner, som vi ej kan implementera med vanlig *call-by-value*. Vilka krav ställer vi i så fall på programmerings-språket? Ge gärna exempel på andra språk där sådana utvidningsmöjligheter finns.

Uppgift 4. Icke-determinism, continuations och logikprogrammering (13p)

Den *icke-deterministiska* amb-evaluatorm för %Scheme implementeras med hjälp av *continuations*. Förklara följande:

4a. (1p) Vad menas med den icke-deterministiska programmeringsparadigmen? Ge exempel på problem som den är lämplig för, utöver det som togs upp på laborationen.

4b. (1p). Man införde bara en ny primitiv amb till %Scheme-språket. Vad gör amb?

```
(amb expr1 expr2 ... exprn)
```

4c. (3p) I amb-evaluatorm "kompileras" eller översätts %Scheme-koden i ett försteg med *analyze* till Scheme-procedurer.

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

Evalueringssmodellen kommer att baseras på *continuations*. Vad innebär detta?

4d. (4p) Primitiven amb "kompileras"/översätts med följande kod:

```
(define (analyze-amb exp)
  (let ((cprocs (map analyze (amb-choices exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices)
             env
             succeed
             (lambda ()
               (try-next (cdr choices))))))
        (try-next cprocs))))))
```

Förklara först vad för slags procedurobjekt, som skapas av ett amb-uttryck. Förklara sedan hur detta objekt används vid evalueringen med *continuations* för att möjliggöra den icke-deterministiska programmeringsmodellen.

4e. (4p) I logikprogrammering används *mönstermatchning* och *unifiering*. Förklara vad dessa begrepp innebär.

Implementeringen av logikprogrammering med QLOG utfördes med hjälp av strömmar. Förklara vad strömmarna innehåller och hur strömmarna skapas och förändras. Till vad användes *mönstermatchning* och *unifiering*?

Uppgift 5. Rekursion (7p)

5a. (2p) Vi har lärt oss att Scheme hanterar *svansrekursion* (tail recursion) 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 svansrekursiv och en som inte är det. Vad kan man vinna genom att hantera svansrekursion på ett eget sätt?

5b. (5p) I den explicita kontrollevaluatorm (bilaga 2) finns kod för *ev-sequence*. Förklara om denna kod ger upphov till en svansrekursiv modell eller ej.

Ge kod för den andra modellen och förklara denna kod.

Uppgift 6. Explicita kontroll-evaluatorm för %Scheme (6p)

Vi vill i %Scheme införa `%and`, som ger ett sant eller falskt värde, och kunna skriva

```
(%and expr1 expr2 ... exprn)
```

Vi vill avsluta beräkningen direkt om något `expri` blir falskt.

Visa hur du implementerar detta i den explicita kontroll-evaluatorm i bilaga 2.

Extrauppgift (med möjlighet till extra poäng max 5p)

På den sista föreläsningen diskuterades *partialevaluering*. Vad innebär det? Det fanns en relation mellan interpretering och kompilering. Kan du förklara detta.

Det finns flera exempel i denna kurs, där partialevaluering skulle kunna vara relevant. Kan du ge exempel och förklara vad vi skulle behöva göra för använda denna teknik.

BILAGA 1

7

;; Core of the evaluator

```
(define (eval exp env) ; motsvarar eval-%scheme
  (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) ; motsvarar apply-%scheme
  (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

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

(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 (reg continue))

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))