Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2011-06-01
Sal (2) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	R41 U15
Tid	8-12
Kurskod	TDDA69
Provkod	TENA
Kursnamn/benämning **Provnamn/benämning**	Data- och programstrukturer Tentamen
Institution	IDA
Antal uppgifter som ingår i tentamen	5
Jour/Kursansvarig Ange vem som besöker salen	Anders Haraldsson
Telefon under skrivtiden	Ankn. 1403, 070-5147709
Besöker salen ca kl.	09,00
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, 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

Onsdag den 1 juni 2011, kl 8-12

Hjälpmedel: Inga.

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

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

Lycka till!!

Anders Haraldsson
Peter Dalenius

Uppgift 1. Beräkningsordning och parameteröverföring (14p)

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

1b. (5p) Vad skiljer parameteröverföringsmodellerna *call-by-name* och *call-by-need*?

Vi har följande uttryck:

```
(define x 100)

(define (f a b c)
  (+ a c c))

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

(f 5 (loop) (begin (set! x (+ x 10)) x))
```

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

a) *call-by-name*
b) *call-by-value*
c) *call-by-need*

Förklara även varför du fick de resultat du angett.

1c. (7p) *Strömmar* (streams) kan inte implementeras med vanlig *call-by-value*. Varför?

Hur löstes det i Scheme för att göra labbarna? I samband med strömmar fanns procedurerna `cons-stream`, `stream-car`, `stream-cdr`, `force` och `delay`.

Du har på labbarna (laboration 3 och 4) i %Scheme implementerat strömmar på 4 olika sätt:

- som special forms
- med makro
- med en lat evaluator
- med att olika parametrar kan överföras med olika överföringsmodeller (`%lazy`, `%lazy-memo`)

Beskriv dessa olika sätt att implementera strömmar och illustrera, om det är lämpligt, med hur du skulle göra det i eval-modellen i bilaga 1.

Uppgift 2. Omgivningsmodellen och statisk & dynamisk bindning/räckbidd (10 p)

2a. (2p) Med hjälp av omgivningsdiagram kan vi förklara olika typer av frågor som vi kan vara intresserade av att få ett svar i ett programmeringsspråk.

Kan en formell parameter ha samma namn som själva proceduren?

```
(define (a a) (+ a 1))
(a 2)

(define (b b) (b 2))
(b a)
```

Motivera svaret med omgivningsdiagram.

2b. (3p) I Scheme (och andra Lisp-dialekter) använder man `let` och `let*` för att introducera lokala variabler. Dessa konstruktioner kan beskrivas i Scheme, som lambda-uttryck. Hur?

Vad händer i följande fall?

```
(define a 10)

(let ((a (+ a 10))
      (b (+ a 20)))
    (+ a b))

(let* ((a (+ a 10))
       (b (+ a 20)))
   (+ a b))
```

Motvivera svaret med omgivningsdiagram.

2c. 5p) *Statsik bindning/räckvidd.* 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 (g h n) (let ((x 5)) (h (+ n x))))

(define (f x) (g (lambda (y) (+ x y)) 6))

(f 1)
```

2d. (3p) Vilket värde skulle vi få om en Lisp-interpretator i uppgift 2c istället använde *dynamisk bindning/räckvidd*?

Beskriv även vad dynamiska bindning/räckvidd är. Vilka ändringar i eval-modellen i bilaga 1 måste vi göra för att få denna bindning/räckvidd?

Uppgift 3. Rekursion (10p)

3a. (3p) I `eval`-definitionen i bilaga 1 där `%Scheme` är implementerad i Scheme är rekursionen i `eval` implicit via Scheme's rekursion. I den explicita kontrollevaluatorn är rekursionen explicit. Förklara hur man implementerar rekursion och hur man ser detta i den explicita kontrollevaluatorn.

3b. (7p) 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?

I den explicita kontrollevaluatorn fanns kod för de båda varianterna. Förklara hur man implementerar svansrekursion. Kan du, med hjälp av ett exempel och nedanstående kod, som beräknar sekvenser, beskriva vad som händer. Vilken ger en svansrekursiv resp. icke svansrekursiv lösning.

```
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-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))
```

Uppgift 4. Logikprogrammering och continuations (12p)

4a. (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
```

4b. (2p) I QLOG användes både *mönstermatchning* och *unifiering*. Vad innebär dessa begrepp och vad användes de till?

4c. (7p) 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* innebär. Vad skiljer det sig från vanlig beräkning av funktionsanrop?

I `amb`-evaluatorn "kompileras" eller översätts %Scheme-koden i ett försteg med `analyze` till Scheme-procedurer.

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

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. 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.

Uppgift 5. Språkutvidgning och evaluatorerna (12p)

Vi önskar i den meta-cirkulära evaluatorn %Scheme implementera proceduren `until`, med syntaxen

`(until` $\texttt{exp}_1$ $\texttt{exp}_2$ `...` $\texttt{exp}_n$ `endtest)`

och med evalueringsordningen av argumenten så att vi först beräknas satserna $\texttt{exp}_i$. Sedan beräknas `endtest`, om det blir sant avslutas repetitionen. Detta repeteras så länge `endtest` är falskt. Som värde returneras symbolen `done`.

Kod för %Scheme med utgångspunkt eval-definitionen finns i bilaga 1. Vi inför dessutom abstraktions-funktionerna:

```
(define (until? exp) (tagged-list? exp 'until))
(define (until-endtest exp) (last-element exp))
(define (repeat-body exp) (cdr (butlast exp)))
```

(Vi antager att `last-element` och `butlast` finns som primitiva funktioner i Scheme)

5a. (4p) Implementera `until` som en *special form* enligt följande

```
((until? exp) (ev-until exp env))
```

Definiera `ev-until`.

5b. (8p) 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)                    ; 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)
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))
```