Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2011-08-18
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	TER4
Tid	14-18
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	0705-147709
Besöker salen ca kl.	15,00 och 17,00
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, ankn. 2362, anna.grabska.eklund@liu.se
Tillåtna hjälpmedel	Inga
Övrigt	
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 18 augusti 2011, kl 14-18

Hjälpmedel: Inga.

Poänggränser: Maximalt kan erhållas 58p. För godkänt krävs ca 24p.

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

Lycka till!!

Anders Haraldsson

Uppgift 1. Beräkningsordning och parameteröverföring (10p)

1a. (2p) Beräkning 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.

Hur är parameteröverföringsmodellerna *call-by-value* och *call-by-name* relaterade till dessa beräkningsordningar?

1b. (4p) Vad skiljer parameteröverföringsmodellerna *call-by-name* och *call-by-need*?

Vi har följande uttryck:

```
(define n 10)

(define (g a b c)
  (if (< a 10) (+ a b) (+ a c c)))

(define (f n) (/ 100 n))

(f (+ n 5) (f 0) (begin (set! n (+ n 1)) n))
```

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

I) *call-by-value*
II) *call-by-name*
III) *call-by-need*

Förklara även varför du fick de resultat du angett.

1c. (4p) I Ada och C har man möjlighet att överföra parametrar med *call-by-reference*. I Ada har vi **out**-deklarerade variabler. Vada innebär *call-by-reference*?

Där kan man definiera en funktion `switch` som byter värdet på två parametrar.

```
(define (switch x y)
  (let ((temp y))
    (set! y x)
    (set! x temp)))
```

Vi kan då byta värden på variablerna `a` och `b`.

```
(define a 10)
(define b 20)
(switch a b)

a = 20
b = 10
```

Kan vi med de någon av de olika modellerna i uppgift 1b definiera denna funktion eller måste vi gå in i eval-modellen. Försök beskriv hur du skulle göra.

Uppgift 2. Omgivningsmodellen och statisk & dynamisk bindning (8p)

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

2a. (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?

2b. (2p) I Scheme (och andra Lisp-dialekter) använder man `let` och `let*` för att introducera lokala variabler. Hur kan dess semantik beskrivas i Scheme. Vad gäller i följande fall? Vilka värden returneras?

```
(define a 10)

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

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

2c. (4p) 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 (f x) (g (lambda (y) (+ x y)) 6))

(define (g h x) (h (+ n x))))

(f 1)
```

2d. (1p) Vilket värde skulle vi få om en Lisp-interpretator i uppgift 2c istället använde *dynamisk bindning*?

Uppgift 3. Objektorientering - procedurobjekt i Scheme (8p)

Vi utgår från följande objektorienterade kod, som hanterar saker och kombinationslås:

```
(define (make-thing name)
  (lambda (msg)
    (case msg
    ((name) name))))

(define (make-container initial-content)
  (let ((content initial-content))
     (lambda (msg)
      (case msg
        ((add) ... se uppgift 3c ...)
        ((get) (lambda (thing)
                 (set! content (delete! thing content))))
        ((content) content)))))

(define (make-safe combo)
  (let ((inner (make-container '()))
        (state 'unlocked))
     (lambda (msg)
      (case msg
        ((unlock) (lambda (input)
                  (if (eq? state 'unlocked)
                    'already-unlocked
                    (if  (eq? input combo)
                      (set! state 'unlocked)
                      'wrong-combo))))
        ((lock) ... på samma sätt som för unlock...)
        ((add) ... se uppgift 3c ...)
        ((get) ... på samma sätt som för add ...)
        ((name-of-content) (if (eq? state 'unlocked)
                               ... se uppgift 3d ...
                               'safe-is-locked)))))
```

3a. (2p) Beskriv delarna i koden i relation till olika begrepp som du finner inom *objektorienterad programmering*. Redogör kortfattat för dessa begrepp.

3b. (2p) Skriv sedan ett Schemeuttryck som gör följande:

1) skapa ett kassaskåp, Kalles kassaskåp med koden `moppe`
2) skapa ett objekt, Kalles ring, med namnet `kalles-ring`
3) lägg in Kalles ring i Kalles kassaskåp
4) skriv ut namnen på de objekt som finns lagrade i Kalles kassaskåp

3c. (2p) Komplettera med kod (märkt ... *se uppgift 3c* ...) för at lägga in ett objekt i ett kassaskåp.

3d. (2p) Komplettera med kod (märkt ... *se uppgift 3d* ...) för att se vad som finns i ett kassaskåp. Vi önskar som resultat en lista med objektens namn.

Uppgift 4. Rekursion (9p)

4a. (3p) Vid implementeringen av %Scheme enligt bilaga 1, finns 2 olika typer av rekursion som skall hanteras, dels användning av implementeringsspråket Scheme's rekursion (t.ex. vi ser de rekursiva anropen av `eval-%scheme`) och dels hantering av rekursion i det implementerade %Scheme-språket.

I den explicita kontrollevaluatorn implementerar vi %Scheme i ett icke-rekursivt språk i form av instruktionerna i registermaskinen (REG). Vi behöver fortfarande göra "rekursiva" anrop till t.ex. `eval-%scheme`. Hur hanteras detta? Visa i koden i bilaga 2.

4b. (4p) 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 svansrekursivt och en som inte är det. Vad kan man vinna genom att hantera svansrekursion på ett eget sätt?

I den explicita kontrollevaluatorn för %Scheme fanns kod för de båda varianterna. Vilken av nedanstående kodavsnitt är svansrekursiv? Kan du, med hjälp av ett exempel, förklara med nedanstående kod vad som händer 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 (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))
```

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 5. Strömmar, logikprogrammering och continuations (10p)

5a. (3p) Vad skilde *strömmar* från listor? Hur implementerades dessa?

5b. (3p) I logikprogrammering finns *mönstermatchning* och *unifiering*. Vad innebär dessa två begrepp?

Logikprogrammeringsspråket QLOG implementerades med hjälp av strömmar. Förklara översiktligt hur detta fungerade och vad de olika strömmarna innehåller för slags dataobjekt. Till vad användes mönstermatchning och till vad användes unifiering?

5c. (4p) 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?

Uppgift 6. Språkutvidgning och evaluatorerna (13p)

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

`(repeat test` $\texttt{exp}_1$ $\texttt{exp}_2$ `...` $\texttt{exp}_n$`)`

och med evalueringsordningen av argumenten så att vi först beräknar `test`. Om värdet är sant beräknas satserna $\texttt{exp}_i$. Detta repeteras tills `test` är falskt. Som värde returneras symbolen `done`.

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

```
(define (repeat? exp) (tagged-list? exp 'repeat))
(define (repeat-test exp) (cadr exp))
(define (repeat-body exp) (cddr expr))
```

6a. (2p) Implementera `repeat` först som en *syntaxutvidgning*, som översätter `repeat`.uttryck till ett `if`-uttryck och sedan evaluerar `if`-uttrycket. Vi lägger in i evaluatorn ett nytt fall

```
((repeat? exp) (eval (repeat->if exp) env))
```

Definiera `repeat->if`.

6b. (2p) Som alternativ kan vi implementera `repeat` som en *special form* enligt följande

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

Definiera `ev-repeat`.

6c. (3p) I uppgift 6a gjorde vi en översättning av ett uttryck. Detta kan generaliseras till *makro-utveckling*, som på samma sätt kan utföras direkt inne i evaluatorn. Antag att våra makroprocedurer har samma form som vanliga procedurer och finns lagrade i omgivningen på samma sätt men är taggade som makroprocedurer. Vi kan testa om en procedur är en makroprocedur med

```
(macro-procedure? proc)
```

Visa med kod var du lägger in makro-utvecklingen i evaluatorn för %Scheme i bilaga 1.

6d. (6p) Implementera proceduren `repeat`, enligt modell 6b, 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)                    ; 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)
```

```
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))
 .....
```