Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2014-10-23
Sal (1)	G34
Tid	14-18
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	Ahmed Rezine
Telefon under skrivtiden	013-281938 eller 0722-031978
Besöker salen ca klockan	ja
Kursadministratör/kontaktperson (namn + tfnr + mailaddress)	Anna Grabska Eklund, ankn. 2362, anna.grabska.eklund@liu.se
Tillåtna hjälpmedel	inga
Övrigt	
Antal exemplar i påsen	

Exam in Data and Program Structure (TDDA69)

Department of Computer and Information Science
Linköping University
2014–10–23
Lecturers: Rezine A., Märak Leffer A.
Time: 14 – –18

Directions:

1. No documents, books or calculators are allowed
2. Write your answers clearly
3. Do not answer to more than one problem on each sheet of paper
4. Do not write on the back of the papers
5. You can answer in English or Swedish
6. Write your identifier on each sheet of paper

Examiner: Ahmed Rezine, 013 28 1938, 072 203 1978

You need about 25 points (out of maximum 50) to pass the exam.

Good Luck!

Problem A. Evaluation order and parameter passing (12 p)

1. (2p) What does it mean for a procedure to be strict in an argument?
2. (2p) Which of the *normal* and *applicative* orders correspond to having procedures strict in all their arguments? explain.
3. (2p) How can memoization speed up lazy evaluation? what is the parameter passing model called?
4. (4p) Write an expression that creates the stream of all natural numbers that are divisible by 3, i.e., *0,3,6,9,12 ...*
5. (2p) Which of the procedures **cons-stream**, **stream-car**, **stream-cdr**, **force** and **delay** should not be evaluated eagerly? explain.

Problem B. The environment model (18 p)

Assume the *environment model* of evaluation.

1. (2p) Use an an *environment diagram* example with several frames and environments to explain the notions of *environments*, *frames*, (shadowed) variables and variables' values.
2. (6p) Assume we pass an expression and an environment **env** to **eval**. Explain how the expression **exp** in **(eval 'exp env)** is evaluated, and what is the result, using an environment diagram (if relevant):
   (a) a self evaluating expression: e.g., **(eval '3 env)**
   (b) a variable name: e.g., **(eval 'num env)**
   (c) a definition: e.g., **(eval '(define par 25) env)**
   (d) an assignment: e.g., **(eval '(set! par 25) env)**
   (e) a lambda expression: e.g., **(eval '(lambda (y z) (/ y z)) env)**
   (f) an application: e.g., **(eval '(bar t) env)**
3. (2p) Explain the semantics of let and let* in Scheme. What is the result of evaluating the code in Fig.1?

```
(define s 1)

(let  ((s (* s 2))(t (* s 2)))
  (* s t))

(let* ((s (* s 2)) (t (* s 2)))
  (* s t))
```

Figur 1: Semantics of **let** and **let***

4. (6p) Assume we evaluate the expressions in Fig.2. What is the result of evaluating the last expression? Draw an environment diagram capturing the most important structures and describe in which order they are created.

```
(define (bar h n) (let ((x 4)) (h (+ n x))))

(define (foo x) (bar (lambda (y) (* x y)) 4))

(foo 2)
```

Figur 2: static vs dynamic binding

5. (2p) What would be the value of the last expression in Fig.2 if the interpreter instead made use of *dynamic binding*?

Problem C. Object oriented programming (4 p)

1. (3p) Describe the code in Fig.3 in terms of object oriented notions.

   What object oriented programming properties can be captured when using the environment model of evaluation?

2. (1p) Define the functions **print-all**, **print-grade** and **set-grade!** in order to allow for a more functional style syntax (described in Fig.5) as opposed to the current one (Fig.4):

```
(define (make-classroom students)

 (define (concat list1 list2)
   (cond ((null? list1) list2)
         (else (cons (car list1) (concat (cdr list1) list2)))))

 (define (find-student name)
   (define (look-in seq)
     (cond ((null? seq) (error "unregistred student"))
           ((eq? name (caar seq)) (list '() (car seq) (cdr seq)))
           (else (let ((found (look-in (cdr seq))))
                   (list (cons (car seq) (car found))
                         (cadr found)
                         (caddr found))))))
   (look-in students))

 (define (set-grade! name grade)
   (let ((found (find-student name)))
     (set! students (concat (car found)
                            (concat (list (list name grade))
                                    (caddr found))))))

 (define (print-grade name) (cadr (find-student name)))

 (define (dispatch m)
   (cond ((eq? m 'set-grade) set-grade!)
         ((eq? m 'print-grade) print-grade)
         ((eq? m 'print-all) (lambda () students))
         (else (error "Unknown request" m))))
 dispatch)

(define class (make-classroom '((Anders 0) (Amy 0) (Erik 0))))
```

Figur 3: capturing object oriented concepts

```
((class 'print-all))
> ((Anders 0) (Amy 0) (Erik 0))
((class 'set-grade) 5)
((class 'print-grade) 'Amy)
> (Amy 5)
```

Figur 4: current syntax

```
(print-all class)
> ((Anders 0) (Amy 0) (Erik 0))
(set-grade! class 'Amy 5)
(print-grade class 'Amy)
>' (Amy 5)
```

Figur 5: targeted syntax

Problem D. Logic Programming and Continuations (8 p)

1. (4p) Define in Prolog or in QLOG[1] a predicate, **palindrome**, that decides whether a list is a plaindrome (i.e., a list $i_1 \ldots i_n$ is a plaindrome iff $\forall k : 1 \leq k \leq n . i_k = i_{n-k+1}$). For instance:

   `(palindrome ()) is true`

[1]Recall the predicate (**append u v w**) that holds exactly when the concatenation **uv** coincides with **w** can be defined in QLOG with the two rules (**rule (append () ?v ?v)**) and (**rule (append (?u . ?v) ?y (?u . ?z)) (append ?v ?y ?z)**)

```
(palindrome (1)) is true
(subset (1 2 1)) is true
(subset (1 1 1 1)) is true
(subset (1 1 2 1)) is false
(subset (1 1 2)) is false
```

2. (4p) In the non-deterministic evaluator (amb-evaluator), the evaluation can "fail". This results in "backtracking" to an earlier choice point. This was implemented using continuations.

   (a) We added a new operation **amb**. Give an expression and its possible evaluations to illustrate the usage of **amb**.

   (b) Explain what a continuation is and how using them implies a different evaluation model compared to the "usual one" (e.g. the evaluation model implemented by %Scheme).

   (c) How is "backtracking" implemented in the amb-evaluator? Explain the role of failure continuations and how to find the following alternative at a previous choice point.

   (d) When backtracking to an earlier choice point, what happens to the side-effects that have already been executed? can they be "undone"?

Problem E. Language extension (8 p)

We would like to implement a procedure "or" such that the evaluation of the arguments stops as soon as one of them evaluates to true. Implement the "or" procedure based on the metacircular evaluator (sketched in the following). You can make use of your own primitives (just explain what they do).

```
(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 ... ))))

(define (apply proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         (eval-sequence (procedure-body proc)
                        (extend-environment
                         (procedure-parameters proc)
                         args
                         (procedure-environment proc))))
        (else (error ... ))))
```

```
(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)

(define (make-procedure param body env)
  ...
```