Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2013-10-24
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	KÅRA
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	072 203 1978
Besöker salen ca kl.	Nej (nås på telefon)
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	Valfritt
Antal exemplar i påsen	

Exam in Data and Program Structure (TDDA69)

Department of Computer and Information Science
Linköping University
2013–10–24
Lecturers: Rezine A., Märak Leffer A.
Time: 14 – –18
Location: Kåra.
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 26 points (out of maximum 52) to pass the exam.

Good Luck!

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

1. (2p) Give an expression whose evaluation differentiates *normal* and *applicative orders* of evaluation.

2. (2p) How are the parameter passing models *call-by-value* and *call-by-name* related to these two evaluation orders?

3. (2p) What is the difference between *call-by-name* and *call-by-need*? explain why debugging can be more difficult in case of lazy evaluation as opposed to the usual *call-by-value* parameter passing model.

4. (4p) Write an expression that creates the stream of all natural even numbers *0,2,4 ...*

5. (2p) Which of the procedures **cons-stream**, **stream-car**, **stream-cdr**, **force** and **delay** cannot be implemented as usual functions? explain.

Problem B. The environment model (18 p)
Assume the *environment model* of evaluation.

1. (2p) In an *environment diagram*, what are *environments*, *frames*, variables and variables' values.

2. (6p) Assume we pass an expression and an environment **env** to **eval**. Explain how the expression is evaluated using (if relevant) an environment diagram:

   (a) a self evaluating expression: e.g., **(eval '1 env)**

   (b) a variable name: e.g., **(eval 'y env)**

   (c) a definition: e.g., **(eval '(define z 1) env)**

   (d) an assignment: e.g., **(eval '(set! x 1) env)**

   (e) a lambda expression: e.g., **(eval '(lambda (x) (- x step)) env)**

   (f) an application: e.g., **(eval '(foo 7) env)**

```
(define x 0)

(let  ((x (+ x 1))
       (y (+ x 1)))
      (* x y))

(let* ((x (+ x 1))
       (y (+ x 1)))
      (* x y))
```

Figur 1: Semantics of let and let*

3. (2p) Explain the semantics of let and let* in Scheme. What is the result of evaluating the code in Fig.1?

4. (2p) Assume we evaluate the expressions in Fig.2.

5. (4p) 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 and disposed of.

```
(define (my-map fn l)
  (if (null? l)
      '()
      (cons (fn (car l))
            (my-map fn (cdr l)))))

(define (foo l)
  (my-map (lambda (x) (cons x l))
          '(a b c)))

(foo '(1 2))
```

Figur 2: static vs dynamic binding

6. (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)

```
(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
```

Figur 3: capturing object oriented concepts

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

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

2. (1p) Define the functions **withdraw!** and **deposit!** in order to allow for a more functional style syntax (described in Fig.5) as opposed to the current one (Fig.4):

```
(define acc (make-account 100))
((acc 'withdraw) 40)
=> 60
((acc 'deposit) 30)
=> 90
```

Figur 4: current syntax

```
(define acc (make-account 100))
(withdraw acc 40)
=> 60
(deposit acc 30)
=> 90
```

Figur 5: targeted syntax

Problem D. Logic Programming and Continuations (8 p)

1. (3p) Define in Prolog or in QLOG a predicate, **subset**, that decides whether a list is a subset of another list.

```
(subset (2 1) (1 2 3)) is true
(subset (1 1 2) (1 2 3)) is true
(subset (2 4) (1 2 3)) is false
```

2. (5p) 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**. What does it do?

   (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** in the appendix).

   (c) How can one "backtrack" in the code? what does a failure-continuation contain? Explain its role 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 (10 p)

1. (4p)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 appendix 1).

2. (6p) Implement the procedure "**or**" in the explicit control evaluator (sketched in appendix 2).

You can make use of your own primitives (just explain what they do).

Appendix 1

```
(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)
  ...
```

Appendix 2

```
eval-dispatch
  (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))

...
```