Försättsblad till skriftlig tentamen vid Linköpings Universitet

Datum för tentamen	2013-06-01
Sal (1) Om tentan går i flera salar ska du bifoga ett försättsblad till varje sal och ringa in vilken sal som avses	TER3
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	Ahmed Rezine
Telefon under skrivtiden	28 1938
Besöker salen ca kl.	-
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	

Exam in Data and Program Structure (TDDA69)

Department of Computer and Information Science
Linköping University
2013–06–01
Lecturers: Rezine A., Märak Leffer A.
Time: 08 – –12
Location: TER3

Directions:

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

Examiner: Ahmed Rezine, 28 1938

You need about 30 points (out of maximum 62) to pass the exam.

Good Luck!

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

1. (2p) Expressions can be evaluated in *normal* or *applicative orders*. Explain both orders. Do they always give the same result? If not, give an example that differentiates them.

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) What is the difference between *streams* and lists? Why is it the case that streams cannot be implemented like other usual functions? Explain the procedures **cons-stream**, **stream-car**, **stream-cdr**, **force** and **delay**.

5. (2p) Write an expression that creates the stream of all natural numbers *0,1,2, ...*

Problem B. The environment model (16 p)
The **eval** procedure in appendix 1, in particular the way variables are bound to their respective values, follows the *environment model* of evaluation.

1. (2p) Describe the elements of an *environment diagram*. What is a *frame*? An *environment*? Where are *variables' values* stored?

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

   (a) a self evaluating expression: e.g., **(eval '0 env)**

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

   (c) a definition: e.g., **(eval '(define n 0) env)**

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

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

   (f) an application: e.g., **(eval '(foo 17 23) env)**

```
(define x 0)

(let  ((x (+ x 10))
       (y (- x 10)))
      (+ x y))

(let* ((x (+ x 10))
       (y (- x 10)))
      (+ x y))
```

Figure 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. (4p) Assume we evaluate the expressions in Fig.2. What is the result of the 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 (g h n) (let ((x 5)) (h (+ n x))))

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

(f 1)
```

Figure 2: static vs dynamic binding

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

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 used the environment model of evaluation?

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

```
(define (make-wire)
 (let ((signal-value 0) (action-procedures '()))
  (define (set-my-signal! new-value)
   (if (not (= signal-value new-value))
     (begin (set! signal-value new-value)
         (call-each action-procedures))
     'done))
  (define (accept-action-procedure! proc)
   (set! action-procedures (mcons proc action-procedures))
   (proc))
  (define (dispatch m)
   (cond ((eq? m 'get-signal) signal-value)
      ((eq? m 'set-signal!) set-my-signal!)
      ((eq? m 'add-action!) accept-action-procedure!)
      (else (error "Unknown operation -- WIRE" m))))
  dispatch))
```

Figure 3: captruting object oriented concepts

```
(define wire (make-wire))

(wire 'get-signal)
=> 0

((wire 'set-signal!) 1)
=> done

((wire 'add-action!) action)
=> ...
```

Figure 4: current syntax

```
(define wire (make-wire))

(get-signal wire)
=> 0

(set-signal! wire 1)
=> done

(add-action! wire action)
=> ...
```

Figure 5: targeted syntax

Problem D. Briefly explain the following concepts. (6 p)

1. referential transparency
2. pattern matching
3. unification
4. continuations
5. garbage collection
6. constraint propagation

Problem E: recursion (8 p)

1. (3p) In appendix 1, the recursion of the **eval** procedure is implicit by using **Scheme**'s recursion. In the explicit control evaluator, this recursion is made explicit. Explain how this recursion is implemented in the second case, and how this is apparent in the explicit control evaluator.

2. (3p) **Scheme** optimizes tail recursion when interpreted. For implementations of other languages this is only performed when the expressions are compiled. What is tail recursion? Give an example of a tail-recursive procedure, and another of a recursive but not-tail recursive procedure. What can be gained by treating tail recursion as a special case?

```
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))
```

Figure 6: version 1 **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))
```

Figure 7: version 2 **ev-sequence**

3. (2p) We define a function **iter** that prints a value every millionth iteration:

```
(define (iter n)
  (cond ((= n 0)
         (begin (display 'ok) (newline)))
        ((= (remainder n 1000000) 0)
         (begin (display n) (newline) (iter (- n 1))))
        (else (iter (- n 1)))))
```

We evaluate **(iter 50000000)**, i.e., **iter** is called 50 millions times.

Is this possible to compute? What is stored in the stack in the tail recursive, respectively non-tail recursive case? Although we created 50 million frames, there is still memory available. How is that possible?

Problem F. Language extension (16 p)
We would like to extend **Scheme** with the **repeat** procedure by modifying the meta-circular evaluator of appendix 1. The syntax is as follows.

```
(repeat test exp1 exp2 ... expn)
```

The evaluation order is as follows: first test is evaluated, if the value is true, the expression exp_i is evaluated. This is repeated until test evaluates to false, or there are no more expressions to evaluate. The value **done** is then returned.
The code for the metacircular evaluator is sketched in appendix 1. We define the following abstraction functions:

```
(define (repeat? exp) (tagged-list? exp 'repeat))

(define (repeat-test exp) (cadr exp))

(define (repeat-body exp) (cddr exp))
```

1. (3p) First implement **repeat** as a syntactic extension that translates **repeat** to an if-expression which is then evaluated. We add to the evaluator the following new case:

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

   Define `repeat->if`

2. (3p) We can also implement **repeat** as follows:

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

   Define `ev-repeat`.

3. (3p) We translated an expression in F:1. This can be generalized to macros expansion. Assume the macro procedures are stored, exactly like usual procedures, in the environment and are tagged as macro procedures. We can test whether a procedure is a macro procedure using:

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

Write commented code that shows where and how you would add macro expansion in the metacircular **Scheme** evaluator of appendix 1.

4. (7p) Implement the **repeat** procedure, according to model F:2, in the explicit control evaluator of appendix 2. You can make use of your own primitive procedures (just explain what they do). You should not assume **eval-repeat** to be a primitive operation.

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))

...
```