

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Fall Semester, 1999

Quiz II

Closed Book – one sheet of notes

Please write clear and concise answers to the questions in the spaces provided in this booklet. While the answers are brief, many of these questions require a lot of thought. You may use scratch paper if you need, but the spaces we provide are the **only** places we will look at when grading.

SOLUTION

Your name:

Circle your *Section Number* and your *Tutor* below:

Section	Time	Location	Rec. Instructor	Tutors
1	10:00	13-4101	Randy Davis	Jim Waldrop
2	11:00	13-4101	Randy Davis	Kyle Ingols
3	10:00	36-155	Daniel Jackson	Jim Waldrop/Mike Phillips
4	11:00	36-156	Daniel Jackson	Robbin Chapman
5	11:00	26-204	Tony Eng	Robbin Chapman/Kyle Ingols
6	12:00	26-204	Tony Eng	Aileen Tang
7	1:00	26-204	Leslie Kaelbling	Mieszko Lis
8	2:00	26-210	Leslie Kaelbling	Mike Phillips
9	12:00	36-153	Tomas Lozano-Perez	Aileen Tang/Mieszko Lis

Please do not write below this line:

Problem	Value	Grade	Grader
1	14		
2	18		
3	20		
4	12		
5	7		
6	12		
Total	83		

Question 1: (14 points)

For each of the following expressions or sequences of expressions, state what value is returned as the result of evaluating the final expression in each set. Each part of this question is independent—in other words, for each part, state what value would be returned if that expression or sequence of expressions were the only expressions evaluated in a newly initialized scheme system.

If the evaluation results in an error, please state in general terms what kind of error. For example, you might write “error: wrong type of argument to procedure.” If the evaluation returns a procedure, simply write the word **procedure**. If the evaluation returns a value that might be different depending on which scheme system you evaluate the expression, write the words **implementation dependent**.

2 points for each answer box.

1. `''a`

`(quote (quote a))`

2. `(+ 5 '(+ 2 3))`

error, wrong type of argument to procedure

3. `(define a 10)`
`(define plus-a (lambda (x) (+ x a)))`
`(set! a 6)`
`(plus-a 5)`

11

4. `(define x '(a b c))`
`(define y (cdr x))`
`(set-car! (cdr x) 'd)`
`(list x y)`

`((a d c) (d c))`

5. `(define a 5)`
`(define fizz`
`(let ((a a))`
`(lambda (x) (+ x a))))`
`(set! a 10)`
`(fizz 2)`

7

```
6. (define b 10)
   (define buzz
     (let ((b b))
       (lambda (x)
         (set! b (+ b x))
         b)))
   (list (buzz 2) b)
```

(12 10)

```
7. (define crackle
     (let ((a 5))
       (lambda (x) (+ x a))))
   ((crackle) 10)
```

error, wrong number of arguments to procedure

Question 2: (18 points)

In this problem, we are interested in a (very!) simple model for how genetic information is passed down from one animal to another. The OOP system from Problem Set 6 is reproduced for you at the end of the exam booklet. There are 3 points for each answer blank in this problem.

We will use a very simple representation for genetic information. The genes will be values for properties such as color and voice. The genes will be represented using a `proplist`, which is a Scheme association list; that is, a list where each member of the list is itself a list associating some `property` symbol with some symbol value. For example:

```
(get-property 'color '((color blue) (voice ribbit)))
;Value: blue
```

where `get-property` is defined as:

```
(define (get-property prop proplist)
  (cond ((null? proplist) #F)
        ((eq? prop (caar proplist))
         (cadar proplist))
        (else (get-property prop (cdr proplist)))))
```

The base class definition for `animal` is the following:

```
(define (make-animal parents genes)
  (lambda (msg)
    (case msg
      ((ANIMAL?) (lambda (self) #T))
      ((GENES) (lambda (self) genes))
      ((CHARACTERISTIC) (lambda (self prop)
                          (get-property prop (ask self 'GENES))))
      ((BIRTH)
       (lambda (self mate)
         (let ((mate-genes (ask mate 'GENES))
               (my-genes (ask self 'GENES)))
           (make-animal (list self mate)
                        (shuffle-genes mate-genes my-genes))))))
      (else (no-method)))))
```

The genetic information for an animal is thus available through the `'GENES` method:

```
(define blue-frog (make-animal nil '((color blue) (voice ribbit))))
(define green-frog (make-animal nil '((color green) (voice bud))))
(ask blue-frog 'GENES)
;Value: ((color blue) (voice ribbit))

(ask green-frog 'GENES)
;Value: ((color green) (voice bud))
```

The interesting behavior comes when we breed two animals. The idea is that their genetic material is somehow mixed together. We will use the procedure `shuffle-genes` to accomplish this:

```
(define (shuffle-genes L1 L2)
  (cond ((null? L1) L2)
        ((null? L2) L1)
        (else (cons (car L1)
                     (shuffle-genes (cdr L2) (cdr L1)))))) ; Note order swap!
```

With the implementation of the `animal` class on the previous page, and the `shuffle-genes` procedure above, consider what happens when we ask an animal to give birth to a new animal in cooperation with some mate:

```
(define new-frog (ask blue-frog 'BIRTH green-frog))
(ask new-frog 'GENES)
==> ???
```

1. What value is returned if you evaluate `(ask new-frog 'GENES)`?

```
((color green) (voice ribbit))
```

Now that we have an `animal` class, it should be possible to create a `dog` class that is a subclass of `animal`. Here is the code to achieve this, with some expressions you are to fill in.

```
(define (make-dog parents genes)
  (let ( <EXP-A> ) ; ; *
    (lambda (msg)
      (case msg
        ((DOG?) (lambda (self) #T))
        ((MAKER-PROC) (lambda (self) make-dog))
        (else <EXP-B> ))))) ; ; *
```

Write the code that goes in place of `<EXP-A>` and `<EXP-B>` so the `dog` class is a subclass of `animal`.

2. `<EXP-A>`

```
(internal-animal (make-animal parents genes))
```

3. `<EXP-B>`

```
(get-method msg internal-animal)
```

In this implementation some unusual behavior occurs. Specifically, consider the following:

```
(define fido (make-dog nil '((voice deep-bark) (hair shaggy) (size big))))

(define daisy (make-dog nil '((voice sharp-bark) (hair brown) (size medium))))

(define rover (ask daisy 'BIRTH fido))

(is-a fido 'animal?)    ==> #T
(is-a fido 'dog?)      ==> #T

(is-a rover 'animal?)  ==> #T
(is-a rover 'dog?)    ==> #F
```

So `rover`, the off-spring of two dogs, is an **animal** but not a **dog**. This is not quite the behavior we desire! To fix this problem, we add the `MAKER-PROC` method to the `dog` class (as shown above). We also modify the `animal` class so that it includes a `MAKER-PROC` method, and change the `BIRTH` method in the `animal` class as follows:

```
(define (make-animal parents genes)
  ...
  ((BIRTH)
   (lambda (self mate)
     (let ((my-genes (ask self 'GENES))
           (mate-genes (ask mate 'GENES))
           (maker-proc <EXP-C>))
       (maker-proc (list self mate)
                   (shuffle-genes mate-genes my-genes)))))) ;corrected
  ((MAKER-PROC) (lambda (self) make-animal))
  (else ...))
```

Write the code that completes the `BIRTH` method above so that `rover` is correctly born as a dog.

4. <EXP-C>

```
(ask self 'MAKER-PROC)
```

We can also add a `cat` class, also a subclass of `animal`, implemented almost identically to the `dog` class:

```
(define (make-cat parents genes)
  (let ((...))
    (lambda (msg)
      (case msg
        ((CAT?) (lambda (self) #T))
        ((MAKER-PROC) (lambda (self) make-cat))
        (else ...))))))
```

Consider what happens when we do the following:

```
(define daisy (make-dog nil '((voice sharp-bark) (hair brown) (size medium))))
(define tomcat (make-cat nil '((voice low-purr) (hair silky) (size small))))

(define catdog (ask daisy 'BIRTH tomcat))
(ask catdog 'GENES)
;Value: ((voice low-purr) (hair brown) (size small))

(is-a catdog 'dog?)
;Value: #t

(is-a catdog 'cat?)
;Value: #f

(ask catdog 'CHARACTERISTIC 'voice)
;Value: low-purr
```

We decide this cross-over of genes from cats to dogs (where `catdog` is a dog but purrs) is somewhat disturbing, and wish to fix our model so this doesn't happen. We modify the dog class as follows:

```
(define (make-dog parents genes)
  (let ((...) ; your previous answer for <EXP-A> inside this let
        (lambda (msg)
          (case msg
            ((DOG?) (lambda (self) #T))
            ((BIRTH)
             (lambda (self mate)
               (if <EXP-D> ; **
                   <EXP-E> ; **
                   (error "no cross-species breeding")))))
            ((MAKER-PROC) (lambda (self) make-dog))
            (else ...)))) ; your previous answer for <EXP-B> inside this else
```

Write the code that completes the `BIRTH` method in the `dog` class above. In each case, the correct answer should be a single line of scheme code.

5. <EXP-D>

```
(is-a mate 'DOG?)
```

6. <EXP-E>

```
(delegate internal-animal self 'BIRTH mate)
```

Question 3: (20 points)

Assume that these expressions are evaluated in sequence in the global environment.

```
(define (f x)
  (if (< x 0)
      (lambda (y) (- y x))
      (lambda (y) (- x y))))
```

```
(define (foo bar x y)
  (let ((g (bar y)))
    (+ (g x) (g y))))
```

```
(foo f 1 -2)
```

Part A. What is the value printed when you evaluate the expression `(foo f 1 -2)`? To solve this, you may want to draw an environment diagram on scrap paper or fill in the diagram on the next page; you will also need the environment diagram later to solve Part B. (2 points)

	3
--	---

Part B. There is an incomplete environment diagram on the next page of the quiz. Your job is to complete it to produce the environment diagram that would be built up during evaluation of the three expressions above, by connecting the arrows to their targets and filling in the values of the variables. Note that you may not need to use all of the frames printed on the page.

Labels of the form *q1* that appear in the diagram are questions for you to answer. Each corresponds to an answer box on this page. Some of the question labels on the diagram are at the tips of arrows; the answers for those questions consist of a single letter between A and K if the arrow connects to one of the lettered arrow heads, the word **GE** if the arrow connects to the global environment, or the word **NONE** if there is no arrow in the environment diagram (for example, if the arrow comes from a frame that was not used).

The remaining question labels on the diagram are inside the frames. The answers for those questions consist of either a number, if the value of the variable is a number, or any of the letter answers or **GE**, if the value of the variable is a pointer, or **NONE**, if the variable has no value in the environment diagram (for example, if the variable is in a frame that was not used).

1 point for each answer.

<p><i>q1.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>GE</td></tr></table></p> <p><i>q2.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>GE</td></tr></table></p> <p><i>q3.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>G</td></tr></table></p> <p><i>q4.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>NONE</td></tr></table></p> <p><i>q5.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>A</td></tr></table></p> <p><i>q6.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>B</td></tr></table></p>	GE	GE	G	NONE	A	B	<p><i>q7.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>GE</td></tr></table></p> <p><i>q8.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>A</td></tr></table></p> <p><i>q9.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>1</td></tr></table></p> <p><i>q10.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>-2</td></tr></table></p> <p><i>q11.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>E</td></tr></table></p> <p><i>q12.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>C</td></tr></table></p>	GE	A	1	-2	E	C	<p><i>q13.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>GE</td></tr></table></p> <p><i>q14.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>-2</td></tr></table></p> <p><i>q15.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>G</td></tr></table></p> <p><i>q16.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>NONE</td></tr></table></p> <p><i>q17.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>G</td></tr></table></p> <p><i>q18.</i> <table border="1" style="width: 100%; text-align: center;"><tr><td>NONE</td></tr></table></p>	GE	-2	G	NONE	G	NONE
GE																				
GE																				
G																				
NONE																				
A																				
B																				
GE																				
A																				
1																				
-2																				
E																				
C																				
GE																				
-2																				
G																				
NONE																				
G																				
NONE																				

ENVIRONMENT DIAGRAM GOES HERE

Question 4: (12 points)

Seth Bang works for NASA and has been asked to implement a `Table` data abstraction with the following interface:

```
make-table: void -> Table<anytype, anytype>  creates a new table
table-put!: Table<k,v>, k, v -> undef         insert a key and value into the table.
                                                replaces previous value for that key, if any
table-get:  Table<k,v>, k -> (v | null)       return value associated with a key, if any
```

He chooses to implement the table using two lists – one for keys and one for values. The i -th value of the values list corresponds to the value associated with i -th key on the key list.

```
(define seth-tag 'twolists)
(define (make-seth-table) (cons seth-tag (cons '() '()) ) )

;; internal selectors
(define (keylist tbl) (cadr tbl))
(define (valuelist tbl) (caddr tbl))

(define (table-get tbl key)
  (define (search keys values)
    (cond ((null? keys) #f)
          ((eq? (car keys) key) (car values))
          (else (search (cdr keys) (cdr values)))))
  (search (keylist tbl) (valuelist tbl)))

(define (table-put! tbl key val)
  < EXP-A >    ;; incorporate key
  < EXP-B >    ;; incorporate val
  'done)
```

Write the code that goes in place of `EXP-A` and `EXP-B`. `table-put!` should insert the new key at the front of the list of keys, and insert the new value at the front of the list of values. The answer to each question is a single scheme expression. 2 points for each answer.

1. <EXP-A>

```
(set-car! (cdr tbl) (cons key (keylist tbl)))
```

2. <EXP-B>

```
(set-cdr! (cdr tbl) (cons val (valuelist tbl)))
```

Several weeks later, to his surprise, Seth notices that his European counterparts have also implemented a `Table` data structure, but they used association lists instead. They fax him their implementation and it looks like the one from lecture:

```
(define table1-tag 'table1)
(define (make-table1) (cons table1-tag nil))
(define (table1-get tbl key)
  (find-assoc key (cdr tbl)))
(define (table1-put! tbl key val)
  (set-cdr! tbl (add-assoc key val (cdr tbl))))
```

The Europeans also let slip another detail about their implementation of association lists (which `find-assoc` and `add-assoc` above manipulate). Their association lists are implemented as

```
association = Pair<key, Pair<value, null>>
association_list = List<association>
```

Seth decides that, in order for the two kinds of tables to interoperate, people will on occasion need to be able to convert one of his tables into one of the European tables. He decides to provide a procedure that converts one of his tables to one of theirs. Complete the procedure `convert` which does this:

```
(define (convert seth-tbl)
  (let ((theirtable (make-table1)))
    (define (enumerate keys values)
      (if (null? keys)
          <EXP-C>
          (begin
             <EXP-D>
             (enumerate <EXP-E> <EXP-F> )))))
      (enumerate (keylist seth-tbl) (valuelist seth-tbl))))
```

Write the code that goes in place of each of the missing expressions in `convert`. 2 points for each answer.

3. <EXP-C>

theirtable

4. <EXP-D>

(table1-put! theirtable (car keys) (car values))
--

5. <EXP-E>

(cdr keys)

6. <EXP-F>

(cdr values)

Question 5: (7 points)

For each statement below, answer whether the statement is **TRUE** or **FALSE**. 1 point for each answer.

1. Copying a copyrighted work (eg, a book or a program) is OK as long as you don't sell the copies (e.g., you can give them away to friends).

FALSE

2. If you can successfully reverse engineer a trade secret (i.e., figure it out from the product), you can legally use that trade secret yourself.

TRUE

3. Computer programs can be copyrighted, and if they qualify, they can also be patented.

TRUE

4. If by (unlikely) chance, two people happen to write the identical program independently, they can both copyright their program.

TRUE

5. The “abstractions test” created by Judge Learned Hand is relevant only to software.

FALSE

6. Anything published on the Web is automatically in the public domain and may be legally downloaded, copied, and used.

FALSE

7. A copyright requires originality, while a patent requires novelty.

TRUE

Question 6: (12 points)

Suppose we want a procedure `prev` with the property that it returns the previous value it was applied to. For example:

```
(prev 1)
; Value: undef
```

```
(prev 2)
; Value: 1
```

```
(prev 3)
; Value: 2
```

Which of the following attempts to implement `prev` works correctly? For each procedure, answer **YES** if the procedure behaves the same as the procedure `prev` described above, or answer **NO** if the procedure does not behave the same as `prev`. You may find that drawing small environment diagrams will help you, but we will only grade the YES/NO answers in the answer box. 2 points for each answer.

1. (define prev1
 (let ((x 'undef))
 (lambda (y)
 (let ((z x))
 (set! x y)
 z))))

YES

2. (define prev2
 (let ((x (list 'undef)))
 (lambda (y)
 (let ((z x))
 (set-car! x y)
 (car z))))

NO

3. (define prev3
 (let ((x (cons 'undef 'undef)))
 (lambda (y)
 (set-car! x (cdr x))
 (set-cdr! x y)
 (car x))))

YES

Suppose we have defined a procedure `prev` that works as described above, and then a single expression is executed.

For each of the following expressions, state what value is returned as the result of evaluating the expression. Each part of this question is independent—in other words, for each part, state what value would be returned if that expression were the only expressions evaluated in a newly initialized scheme system with the `prev` procedure defined.

If the evaluation results in an error, please state in general terms what kind of error. For example, you might write “error: wrong type of argument to procedure.” If the evaluation returns a procedure, simply write the word **procedure**. If the evaluation returns a value that might be different depending on which scheme system you evaluate the expression, write the words **implementation dependent**.

2 points for each answer.

4. `(let ((x (prev prev)))
 ((prev +) prev) 1 2))`

3

5. `(let ((foo (prev prev)))
 (((foo +) foo) 1 2))`

error, operator not a procedure

6. `(let ((g (prev prev))
 (f (prev prev)))
 (((f +) f) 1 2))`

implementation dependent

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS**OOB System (same as in Problem Set 6)**

```

(define (ask object message . args)
  (apply-method object object message args))

(define (delegate to from message . args)
  (apply-method to from message args))

(define (apply-method in-object for-object message args)
  (let ((method (get-method message in-object)))
    (cond ((method? method)
           (apply method for-object args))
          ((eq? in-object for-object)
           (error "No method for" message 'in (ask in-object 'NAME)))
          (else (error "Can't delegate" message
                       "from" (ask for-object 'NAME)
                       "to" (ask in-object 'NAME))))))

(define (get-method message object) ; single-inheritance
  (object message))

(define (find-method message . objects) ; multiple-inheritance
  (define (try objects)
    (if (null? objects)
        (no-method)
        (let ((method (get-method message (car objects))))
          (if (not (eq? method (no-method)))
              method
              (try (cdr objects))))))
  (try objects))

(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else (error "Object returned this non-message:" x))))

(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))

(define (is-a object type-pred)
  (and (procedure? object)
       (let ((method (get-method type-pred object)))
         (if (method? method)
             (ask object type-pred)
             #F))))

(define (creator maker)
  (lambda args ;weird syntax for proc that takes 0 or more args
    (let ((object (apply maker args)))
      (ask object 'INSTALL)
      object)))

(define (names-of objects)
  (map (lambda (x) (ask x 'NAME)) objects))

```

List Procedures

```
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst) (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

```
(define (map op lst)
  (if (null? lst)
      nil
      (cons (op (car lst))
            (map op (cdr lst)))))
```

```
(define (accumulate op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (accumulate op init (cdr lst)))))
```

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS