

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Spring Semester, 2006

Final Exam – Solutions

Throughout this exam, we have set aside space in which you should write your answers. Please try to put all of your answers in the designated spaces, as we will look only in this spaces when grading.

Note that any procedures or code fragments that you write will be judged not only on correct function, but also on clarity and good programming practice. Also note that while there may be a lot of reading to do on a problem, there is relatively little code to write, so please take the time to read each problem carefully.

NAME:

Section Time:

Tutor's Name:

PART	Value	Grade	PART	Value	Grade
1	20		6	15	
2	18		7	25	
3	20		8	25	
4	22		9	15	
5	20		10	20	
			Total	200	

No grades for 6.001 will be available until May 29th. After that time, you may (1) wait until grades are mailed by the Registrar; (2) access your grades via WEBSIS (<http://student.mit.edu/>), or (3) contact the course secretary, Donna Kaufman, in 38-401. No grades will be given out by phone; you must appear in person with ID if selecting the third option.

Completed final exams will not be handed back to students, but will be available starting on May 29th for students to look at (but not remove from) Donna Kaufman's office, Room 38-401.

Part 1 (20 points): Orders of growth

Question 1: – 4 points

For the procedure `split`, what are the orders of growth?

linear, constant

Question 2: – 4 points

For the procedure `merge`, what are the orders of growth?

linear, linear

Question 3: – 12 points

infinite loop

correct

unsorted

Part 2 (18 points): objects with state

Question 4: What is the sequence of values returned for the following expressions? If one of the expressions causes an evaluation error, write “error” for that expression and don’t worry about the value for later expressions. List the values for all of the expressions, in order, not just the value of the last expression.

nothing, 0, 0, 0

3 points

Question 5:

nothing, 0, 1, 2

3 points

Question 6:

error

3 points

Question 7:

Define a function `fib` that creates an object that generates the fibonacci sequence. For example, we should be able to do the following:

```
(define z (fib))
(z) ;-> returns the value 1
(z) ;-> returns the value 1
(z) ;-> returns the value 2
(z) ;-> returns the value 3
(z) ;-> returns the value 5
```

```
(define (fib)
  (let ((f1 1)
        (f2 1))
    (lambda ()
      (let ((val f1)
            (sum (+ f1 f2)))
        (set! f1 f2)
        (set! f2 sum)
        val))))
```

Grading scheme:

- return lambda of no args – 2 points
- capture state locally – 3 points
- update state – 2 points
- return right value – 2 points

Part 3 (20 points): List structures:**Question 8:** Define a `subsets` procedure that returns all subsets of a given list.

```
(define (subsets l)
  (if (null? l)
      '()
      (let ((tail-subsets (subsets (cdr l))))
        (append (map (lambda (xs) (cons (car l) xs)) tail-subsets)
                tail-subsets))))
```

Grading scheme – 10 points:

- test for base case – 1
- returning correct value – 3 (1 if they just return empty list)
- recursive step – 6

Question 9: Using `list-ref`, write a procedure for `tree-ref`.

```
(define (tree-ref tree indices)
  (if (null? indices)
      tree
      (tree-ref (list-ref tree (car indices))
                (cdr indices))))
```

Grading scheme – 6 points:

- test – 1
- base case – 1
- recursive step – 4

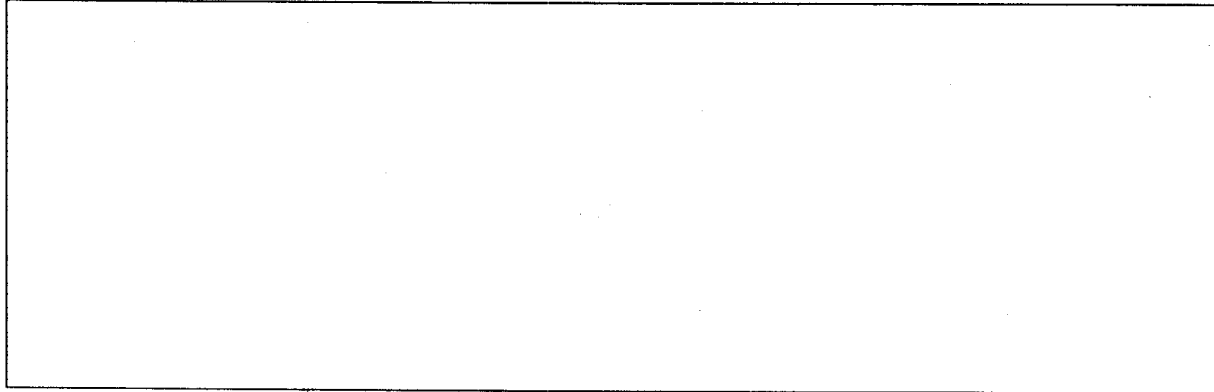
Question 10:

```
(define (tree-ref tree indices)
  (fold2 (LAMBDA (X Y) (LIST-REF X (CAR Y))) tree indices))
```

4 points

Part 4 (22 points): Evaluators**Question 11: – 3 points**

As practice for manipulating the new special form, draw the box-and-pointer diagram for expression 2 above.

**Question 12: – 5 points**

```
(and (tagged-list? exp 'if)
     (= (length exp) 6)
     (eq? (list-ref exp 2) 'then)
     (eq? (list-ref exp 4) 'else))
```

Question 13: – 6 points

```
(let ((pred (list-ref exp 1))
      (conseq (list-ref exp 3))
      (alt (list-ref exp 5)))
    (if (m-eval pred env)
        (m-eval conseq env)
        (m-eval alt env)))
```

Question 14: – 8 points

```
(let ((pred (list-ref exp 1))
      (conseq (list-ref exp 3))
      (alt (if (> (length exp) 4) (list-ref exp 5) 'false)))
    (if (m-eval pred env)
        (m-eval conseq env)
        (m-eval alt env)))
```

Part 5 (20 points): syntactic transforms

Question 15: We are going to write the procedure `for->if`. First we want a procedure that creates a `for` expression:

```
(define (make-for var init end body)
  YOURANSWER)

(append (list 'for var init end)
        body)
```

grading – 2 points, 1 for basic list, 1 for attaching body correctly

Here is a template for the syntactic transformation. The basic idea is that we are going to create a local frame using a `let`, in which we can bind the loop variable, and relative to which we can evaluate the subsequent expressions.

```
(define (for->if exp)
  (list 'let
        ANSWER16
        (list 'if
              ANSWER17
              ''done
              ANSWER18)))
```

Question 16: The expression for `ANSWER16` should create an expression that when evaluated will bind the variable to the initial value. Provide that expression.

```
(list (list (for-var exp) (for-start-value exp)))
```

Grading scheme – 5 points:

- top level list – 2
- interior list – 3

Question 17: The expression for `ANSWER17` should create an expression that when evaluated will determine if the `for` should be exited. Provide that expression.

```
(list '> (for-var exp) (for-end-value exp))
```

2 points

Question 18: The expression for `ANSWER18` should create an expression that when evaluated will evaluate the body of the `for` expression and then evaluate a new `for` expression in an iterative fashion. Provide that expression. Be sure to use `make-for` where appropriate.

```
(append (cons 'begin (for-body exp))
        (list (make-for (for-var exp) (+ 1 (for-start-value exp))
                        (for-end-value exp) (for-body exp))))
```

Grading scheme – 11 points:

- use of begin – 2
- including body – 2
- correct new for structure – 3
- gluing together properly – 4

Part 6 (15 points): environment diagrams

Attached to this exam is an environment diagram, which is the result of evaluation of a single expression. What is that expression? Note that implied procedure bubbles from a `let` may not be shown. Assume that the top level environment is the global environment.

```
(define test (let ((init 5)
                  (bar (lambda (x) (* x x))))
              (lambda (y) (bar (* 2 y)))))
```

Part 7 (25 points): Object-oriented systems

This problem explores a small object-oriented world. The desired behavior of the classes is:

- thing is an object with a name
- person is a subclasses of thing
- an authored-thing is an object with an author, who is a person
- book is a subclass of thing and of authored-thing; it has a genre

The code defining the classes is shown below. (Note: in the code below, methods names are uppercase; in your answers you may use either upper case or lower case.)

```
(define (thing self name)
  (let ((root-part (make-root-object self)))
    (make-handler
     'thing
     (make-methods
      'NAME (lambda () name))
     root-part)))

(define (authored-thing self person)
  (let ((root-part (make-root-object self)))
    (make-handler
     'thing
     (make-methods
      'AUTHOR (lambda () person))
     root-part)))

(define (person self name)
  (let ((thing-part (make-thing self name)))
    (make-handler
     'person
     (make-methods
      '( ))
     thing-part)))

(define (book self name author genre)
  (let ((thing-part (make-thing self name))
        (authored-thing-part (make-authored-thing self author)))
    (make-handler
     'book
     (make-methods
      'GENRE (lambda () genre)
      'AUTHOR-NAME ANSWER-4)
     thing-part authored-thing-part)))
```

```

(define (create-person name)
  (make-instance person name))

(define (create-book name author genre)
  (make-instance book name author genre))

(define book-data
  ;; list of 3-element sublists, each of which has name, author name, genre
  '(("The Diamond Age" "Neal Stephenson" science-fiction)
    ("Neuromancer" "William Gibson" science-fiction)
    ("Player Of Games" "Iain Banks" science-fiction)
    ("Shrub: The Short But Happy Political Life Of George W. Bush" "Molly Ivins" political-satire)))

(define book1
  (create-book "Ubik" (create-person "Philip K. Dick") 'science-fiction))

```

Question 19:

What is the value returned by the expression: `(equal? (ask book1 'AUTHOR) "Philip K. Dick")`?

2

#f

Question 20:

```

(define (create-books book-data)
  ANSWER-20)

```

Fill in the code for a procedure that takes a list of book data, such as `book-data` defined above, and returns a list of book objects, one book object per 3-element sublist in the book data list.

5

```

(map (lambda (x)
      (create-book (car x) (create-person (cadr x)) (caddr x)))
     book-data)

```

no create-person -1
 λ with 3 args -2
 only creates 1st book -3

Question 21:

```
(define (book self name author genre)
  (let ((thing-part (make-thing self name))
        (authored-thing-part (make-authored-thing self author)))
    (make-handler
     'book
     (make-methods
      'GENRE (lambda () genre)
      'AUTHOR-NAME ANSWER-21))
    thing-part authored-thing-part))))
```

Provide the code for ANSWER-21, which returns the name of a book's author.

5

```
(lambda ()
  (ask (ask self 'AUTHOR) 'NAME))
or (ask author 'NAME)
```

no λ -1
no (ask author 'NAME) -2

Question 22:

Fill in the code for a procedure that returns all the books of a particular genre in a list of books.

```
(define (find-it-by-genre list-of-books genre)
  ANSWER-22)
```

5

```
(filter (lambda (x) (eq? (ask x 'GENRE) genre))
  list-of-books)
```

map instead of filter -2

no ask GENRE -2

no list-of-books -1

not eq? instead of eq? -2

Question 23 You decide to add DESCRIPTION methods to your object definitions:

```
(define (thing self name)
  ...
  (make-methods
    'NAME (lambda () name)
    'DESCRIPTION (lambda () (append '(a thing named) (ask self 'NAME)))
    'NAME
    ...)))
```

(list
↑

```
(define (authored-thing self person)
  ...
  (make-methods
    'AUTHOR (lambda () person)
    'DESCRIPTION (lambda () (append '(an authored-thing written by)
                                     ((ask self 'AUTHOR-NAME))
                                     (ask self 'AUTHOR-NAME)))
    ...)))
```

```
(define (book self name author genre)
  ...
  (make-methods
    'GENRE (lambda () genre)
    'DESCRIPTION (lambda ()
                  (append '(a book named) (ask self 'NAME))
                  (written by) (ask self 'AUTHOR-NAME))
    ...)))
```

(list
↑

Recall that book1 was defined as:

```
(define book1
  (create-book 'Ubik' (create-person 'Philip K. Dick')
    'science-fiction))
```

What is returned as the value of (ask book1 'DESCRIPTION)?

Select from:

- '(a thing named 'Ubik')
- '(a book named 'Ubik')
- '(an authored-thing written by 'Philip K. Dick')
- '(a book written by 'Philip K. Dick')
- '(a book named 'Ubik' written by 'Philip K. Dick')
- '(a book written by 'Philip K. Dick' named 'Ubik')

3

Question 24 You decide that a better way to construct a description of a book is to append partial descriptions of book's superclasses. You define the following 'WHAT methods:

```
(define (thing self name)
  ...
  (make-methods
   'NAME (lambda () name)
   'WHAT (lambda () (append list (ask self 'NAME))
   ...)))

(define (authored-thing self person)
  ...
  (make-methods
   'AUTHOR (lambda () person)
   'WHAT (lambda () (append '(written by) (list (ask self 'AUTHOR-NAME)))
   'AUTHOR-NAME)
   ...)))

(define (book self name author genre)
  ...
  (make-methods
   'GENRE (lambda () genre)
   'DESCRIPTION (lambda () ANSWER-24)
   ...)))
```

You'd like a book description of the form:

```
'(a <genre> book entitled <name> written by <author-name>)
```

The description of book1, for example, would be:

```
'(a science-fiction book entitled "Ubik" written by "Philip K. Dick")
```

Complete the definition of book's DESCRIPTION method by writing code for ANSWER-24 that calls superclass WHAT methods.

5

```
(append '(a) (list genre 'book 'entitled)
         (ask thing-part 'WHAT)      or (ask self 'WHAT)
         (ask authored-thing-part 'WHAT))

or '(a ,genre book entitled ,@(ask thing-part 'WHAT)
    ,@(ask authored-thing-part 'WHAT))
```

Part 8 (25 points): higher order procedures**Question 23:**

```
(lambda (x) (/ (- (f (+ x delta-x)) (f (- x delta-x)))
              (* 2 delta-x)))
```

grading scheme – 5 points; 2 for lambda, 3 for algebra

Question 24: What expression should be used for ANSWER24, in order to implement a procedure that computes the n^{th} derivative?

f

3 points

Question 25: What expression should be used for ANSWER25?

```
(deriv (nth-deriv f (- n 1)))
```

grading scheme – 5 points; 2 for deriv, 3 for nth-deriv

Question 26: Finally, provide the procedure `sumseries`. This should sum up the terms in the Taylor series expression, up to the n^{th} term.

```
(define (sumseries f x k n)
  (if (> k n)
      0
      (+ (* ((nth-deriv f k) 0)
            (expt x k)
            (/ 1 (fact k)))
         (sumseries f x (+ k 1) n))))
```

grading scheme – 12 points

- test – 2
- base case – 2
- recursive call – 2
- right use of `expt` and `fact` – 2
- right use of `nth-deriv` – 4

Part 9: (15 points): streams**Question 27:**

```
(define (alternate-streams s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                    (alternate-streams s2 (stream-cdr s1)))))
```

Grading – 6 points

OR

```
(define (alternate-streams s1 s2)
  (cons-stream (stream-car s1)
               (cons-stream (stream-car s2)
                             (alternate-streams (stream-cdr s1)
                                                 (stream-cdr s2)))))
```

Question 28:

```
(define (alternate-list-of-streams l-of-ss)
  (cond ((null? l-of-ss) null-stream)
        ((null? (cdr l-of-ss)) (car l-of-ss))
        (else (alternate-streams
                (car l-of-ss)
                (alternate-list-of-streams (cdr l-of-ss))))))
```

Grading – 9 points; 3 for each base case, 3 for recursive call

Part 10 (20 points): models of evaluation

Question 29:

What is the value of the statement

`(initialized-list (accum) 5)`

in the ordinary (applicative order) evaluator?

(15 14 12 9 5) – 4 points

What about in the lazy, non-memoized evaluator?

(1 2 3 4 5) – 4 points

Question 30:

Suppose that `proc1` and `proc2` are being applied asynchronously, with no serialization. What is the set of possible values for `x` at the completion of both evaluations?

3, 4, 5, 6, 9. – 2 points

Question 31: Suppose that the `set!` expressions in both `proc1` and `proc2` in the previous question are serialized by the same serializer, and that the two procedures are being applied asynchronously. What is the set of possible values for `x` at the completion of both evaluations?

5 or 9. – 2 points

Question 32: – 1 point each

Answer – 0

Answer – 1

Answer – 1

Answer – ~~3~~ 5

Question 33: – 4 points What is the value of the final expression?

Answer – ((20) (30 20))