

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2007

Quiz II – Solutions

Closed Book – two sheets of notes

Throughout this quiz, 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 space 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	Grader
1	28		
2	20		
3	27		
4	25		
Total	100		

For your reference, the TAs are:

- Matt Brown
- Austin Clement
- Michael Craig
- Stephen McCamant
- Brennan Sherry
- James Wnorowski
- Sarah Wu

Part 1. (28 points)

Consider the following sequence of expressions:

```
(define (make-switch initial)
  (let ((state initial))
    (lambda ()
      (let ((old state))
        (if (eq? state 'off)
            (set! state 'on)
            (set! state 'off))
          old))))

(define wall-plate (make-switch 'off))

(wall-plate)
```

Suppose we trace out the evaluation of these expressions with an environment model. Below is a partially completed environment diagram. Note that each procedure and each frame is labeled, however these labels **do not** reflect the order in which these elements were generated. You are to use these labels to answer the questions below; specifically you should complete this diagram, and then use the labels on the parts of the environment diagram to answer the following questions. Remember that `let` is syntactic sugar for a procedure application.

Question 1: For each of the following frames, indicate the lowest frame of the enclosing environment, choosing one of **GE**, **E1**, **E2**, **E3**, **E4** or **none** or **not shown**.

Frame:	Enclosing Environment
GE	none
E1	E4
E2	GE
E3	E1
E4	E2

Question 2: For each of the following procedure objects, indicate the lowest frame of the environment to which the procedure's environment pointer points, choosing one of **GE**, **E1**, **E2**, **E3**, **E4** or **none** or **not shown**.

Procedure:	Procedure's Environment Pointer
P1	GE
P2	E1
P3	E2
P4	E4

Question 3: For each of the following variable names, indicate the value to which it is bound in the specified environment at the **end** of the evaluation of the expressions. Indicate the value by choosing one of **GE**, **E1**, **E2**, **E3**, **E4**, **E5** or **P1**, **P2**, **P3**, **P4** or a symbol, a number, a list of numbers or a boolean value.

Variable:	Environment	Value
make-switch	GE	P1
wall-plate	GE	P4
initial	E2	off
old	E3	off
state	E4	on

Part 2: (20 points)

We have seen in lecture that searching a tree structure is an important task in many applications. For this part of the quiz, we are going to search in a tree for the first node that satisfies a predicate `reach-goal?`. All we need to know about the tree abstraction, in addition to `reach-goal?`, is that it includes a selector `children`, which when applied to a node in the tree returns a **list** of that node's children.

Here is a tree search procedure:

```
(define (search start reach-goal? combine)
  (define (search1 queue)
    (if (empty-queue? queue)
        #f
        (let ((current (front-queue queue)))
          (if (reach-goal? current)
              current
              (search1 (list-to-queue
                        (combine
                         (children current)
                         (queue-to-list (rest-queue queue))))))))))
  (search1 (list-to-queue (list start))))
```

Question 4: For a depth first search, what expression can be used in place of `combine` in the procedure `search`?

`append`

Question 5: For a breadth first search, what expression can be used in place of `combine` in the procedure `search`?

```
(lambda (x y) (append y x))
```

Question 6: For a best first search, what expression can be used in place of `combine` in the procedure `search`?

```
(lambda (x y) (sort (append x y)))
```

Now, let's implement the queue abstraction. We will use a tagged data structure, and will use a cons pair to point to the front and rear of the queue.

```
(define (list-to-queue lst)
  (cons 'queue (cons lst
                     (if (null? lst) '() (last-pair lst)))))

(define (last-pair lst)
  (if (null? (cdr lst))
      lst
      (last-pair (cdr lst))))
```

Question 7:

Provide definitions for each of the following procedures:

```
(define front-queue caadr)

(define (rest-queue q)
  (if (null? (front-queue q))
      (error "can't do this")
      (cons 'queue
            (cons (cdadr q) (cddr q)))))

(define queue-to-list cadr)
```

OR

```
(define queue-to-list front-queue)
```

Part 3: (27 points)

We are going to build a new kind of data structure, called a doubly linked list, or a `dlist`. This is a data structure that has the property that we can in constant time reach the next element in the structure as well as the previous element in the structure. We are going to build `dlists` out of lists, and an example is shown in the figure below:

We define the following operations on `dlists`.

```
(define empty-dlist '())

(define (empty-dlist? obj)
  (null? obj))

(define (value dlist)
  (if (empty-dlist? dlist)
      (error "Cannot take value of an empty dlist")
      (caar dlist)))

(define (next dlist)
  (if (empty-dlist? dlist)
      (error "Cannot take next of an empty dlist")
      (cdr dlist)))

(define (previous dlist)
  (if (empty-dlist? dlist)
      (error "Cannot take previous of an empty dlist")
      (cdar dlist)))
```

Question 8: To complete this abstraction, we need to provide procedures that can mutate the `dlist` to change the `next` and `previous` parts of a `dlist`. Note that a `successor` or a `predecessor` will be a `dlist`.

Provide the missing code for the procedures below:

```
(define (set-next! dlist successor)
  (if (empty-dlist? dlist)
      (error "Cannot set next of an empty dlist")
      ANSWER8-A)
  dlist)

(define (set-previous! dlist predecessor)
  (if (empty-dlist? dlist)
      (error "Cannot set previous of an empty dlist")
      ANSWER8-B)
  dlist)
```

ANSWER8-A

```
(set-cdr! dlist successor)
```

ANSWER8-B

```
(set-cdr! (car dlist) predecessor)
```

Next, we need procedures that create `dlists`. You may assume that your implementation for `set-next!` and `set-previous!` work correctly.

Question 9: The procedure `prepend!` takes as argument a value and a `dlist`, and returns a new `dlist` with the value as the first element, followed by the original `dlist`.

```
(define (prepend! item dlist)
  ANSWER9)
```

ANSWER9:

```
(define (prepend! item dlist)
  (if (empty-dlist? dlist)
      (list (list item))
      (let ((new (cons (list item) dlist)))
        (set-previous! dlist new)
        new)))
```

Question 10: The procedure `postpend!` should take two `dlists` as input, and should return a mutated version of the first `dlist`, in which the second `dlist` immediately follows the last element of the first `dlist`.

```
(define (postpend! dlist1 dlist2)
  (if (empty-dlist? dlist1)
      dlist2
      (let ((end-dlist1 (last-pair dlist1)))
        ANSWER10
        )))
```

```
(define (postpend! dlist1 dlist2)
  (if (empty-dlist? dlist1)
      dlist2
      (let ((end-dlist1 (last-pair dlist1)))
        (SET-NEXT! END-DLIST1 DLIST2)
        (IF (NOT (EMPTY-DLIST? DLIST2))
            (SET-PREVIOUS! DLIST2 END-DLIST1))
        DLIST1)))
```

Question 11: Once we have a `dlist`, we would like to be able to apply a procedure to each element, much as `map` creates a new list out of an old one. Our new procedure, `dmap!`, takes as argument a procedure to apply, and a `dlist`.

```
(define (dmap! proc dlist)
  (for-each-forward proc dlist)
  (if (not (null? (previous dlist)))
      (for-each-reverse proc (previous dlist)))
  dlist)
```

The procedure `for-each-forward` will modify every element in the `dlist`, starting at the current one, and moving in the forward direction along the `dlist`. It takes advantage of the fact that a `dlist` looks like a list. Remember that `for-each` is like `map` except it doesn't collect a list of the results returned by the procedure, it simply applies the procedure to each element of the list for its effect, and throws away the returned values.

Write `for-each-forward`:

```
(define (for-each-forward proc dlist)
  (for-each ANSWER11-A
            dlist))
```

ANSWER11-A

```
(lambda (x) (set-car! x (proc (car x))))
```

The procedure `for-each-reverse` will modify every element in the `dlist`, starting at the current one, and moving in the reverse direction along the `dlist`.

Write `for-each-reverse`:

```
(define (for-each-reverse proc dlist)
  (if (not (empty-dlist? dlist))
      ANSWER11-B))
```

ANSWER11-B:

```
(begin (set-car! (car dlist) (proc (value dlist)))
       (for-each-reverse proc (previous dlist)))
```

Part 4. (25 points)

In this problem, we are going to construct a *time machine procedure* that remembers the argument values it was passed. Whenever a time machine procedure is called, it stores the value passed on the current call, and returns the value that was passed n calls earlier, where n is a number provided when the time machine was constructed. If the time machine is new, so that fewer than n calls have been made to it, then it returns an initial value provided when the time machine was constructed.

The procedure `(make-time-machine n initial-value)` creates a time machine that delays values for n calls, with an initial value `initial-value`. For example:

```
(define tm1 (make-time-machine 1 #f))
(tm1 5) => #f
(tm1 'a) => 5
(tm1 8) => a
```

```
(define tm0 (make-time-machine 0 #f))
(tm0 5) => 5
(tm0 'a) => a
(tm0 8) => 8
```

```
(define tm2 (make-time-machine 2 #f))
(tm2 5) => #f
(tm2 'a) => #f
(tm2 8) => 5
```

We are going to make a time machine two different ways.

Question 12: One way to create a time machine is to maintain an internal state variable that keeps track of all of the arguments passed in to the procedure, and returns the n^{th} previous one, provided there have been sufficient previous calls of the procedure.

Using this idea, implement `make-time-machine`:

```
(define (make-time-machine n initial-value)
  (let ((history '()))
    (lambda (value)
      (set! history (cons value history))
      (if (>= n (length history))
          initial-value
          (list-ref history n))))))
```

There is a different way of implementing this idea, however. Delaying for n calls means the time machine must store up to n values waiting to be returned. In this implementation, however, a time machine should store only *one* value in itself, and use a smaller time machine to remember the remaining $n - 1$ values.

Question 13: Write a definition of `make-time-machine` that gives rise to a recursive process by filling in ANSWER13-A and ANSWER13-B below.

```
(define (make-time-machine n initial-value)
  (if (= n 0)
      ANSWER13-A
      (let ((time-machine (make-time-machine (- n 1) initial-value)))
        ANSWER13-B)))
```

Your answers should use expressions from the list below. Not all the expressions in the list are useful or correct.

```
(lambda (value) ...)
(let ((value-to-return (time-machine saved-value))) ...)
(let ((saved-value initial-value)) ...)
(let ((time-machine (make-time-machine (- n 1) initial-value))) ...)
(set! saved-value value)
(set! returned-value saved-value)
saved-value
value
value-to-return
```

ANSWER13-A (hint: think about what should happen if there is no delay):

```
(lambda (x) x)
```

ANSWER13-B (hint: think about how you can use a time machine with a shorter delay to compute a return value):

```
(define (make-time-machine n initial-value)
  (if (= n 0)
      (lambda (value) value)
      (LET ((TIME-MACHINE (MAKE-TIME-MACHINE (- N 1) INITIAL-VALUE)))
        (LAMBDA (VALUE)
          (LET ((VALUE-TO-RETURN (TIME-MACHINE INITIAL-VALUE)))
            (SET! INITIAL-VALUE VALUE)
            VALUE-TO-RETURN))))))
```