

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring Semester, 1998, Final Exam Solutions

Your Name:

Below are suggested answers for each problem, though in many cases there were alternative answers.

**Question 1 (12 points):**

**Part a:** 25

**Part b:** 20

**Part c:** ((x y x z) x z)

**Question 2 (20 points):**

The correct answers for the true/false questions are:

T F T F F T F F F F

**Question 3 (21 points):**

**Part a:** The data abstraction for dealing with `until` expressions:

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

and the selectors

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

```
(define (until-test exp)
  (cadr exp))
```

**Part b:** The following dispatch should be added after the primitive expressions and before the application in `eval`:

```
((until? exp)
 (eval-until exp env))
```

**Part c:** The procedure `eval-until`:

```
(define (eval-until exp env)
  (let ((return (eval-sequence (until-body exp) env)))
    (if (true? (eval (until-test exp) env))
        return
        (eval-until exp env))))
```

#### Question 4 (14 points):

The code to add the `until` special form of Question 3 to the Explicit-Control Evaluator:

```
ev-until
  (save continue)
  (save exp) ; save the entire expression
  (assign unev (op until-test) (reg exp)) ; get the end test
  (save unev) ; save it for later
  (assign unev (op until-body) (reg exp)) ; get the body
  (assign continue (label ev-after-until-body)) ; where to go when done
  (save env) ; save the environment
  (save continue)
  (goto (label ev-sequence)) ; evaluate the body

ev-after-until-body ; now need to evaluate the end test
  (restore env)
  (restore unev)
  (assign exp (reg unev))
  (save val)
  (save env)
  (assign continue (label ev-after-until-test))
  (goto (label eval-dispatch)) ; evaluate the end test

ev-after-until-test ; check to see if end test is true
  (restore env)
  (test (op true?) (reg val))
  (branch (label done-until)) ; branch if end test true
  (restore val)
  (restore exp)
  (restore continue)
  (goto (label ev-until))

done-until ; clean up when done
  (restore val)
  (restore exp)
  (restore continue)
  (goto (reg continue))
```

#### Question 5 (14 points):

**Part a:** `Arg1` will contain the value of:

```
(list 3)
```

**Part b:** Lines 13 to 24 are created by compiling:

```
(g y)
```

**Part c:** After lines 25 to 27, plus lines 9 to 12, `arg1` will contain:

```
(list 9 3)
```

**Part d:** After lines 25 to 29, plus lines 9 to 12, `arg1` will contain:

```
(list 27 9 3)
```

**Part e:** Lines 6 to 39 are created by compiling:

```
(f x (g y) 3)
```

**Part f:** A Scheme expression whose compilation would produce the entire code:

```
(define (doit x y f g) (f x (g y) 3))
```

### Question 6 (22 points):

**Part a:** The environment diagram should have two frames, one from the application of the procedure and one from the internal `let`. `Trial` should point to a procedure object whose environment pointer points to the chain of frames starting with that created by the `let`.

**Part b:** The definition of `connect`:

```
(define (connect from to)
  ((from 'set-next) to)
  ((to 'set-previous) next))
```

The full definition of `ripple`:

```
(define (ripple new current)
  ;; idea is to insert in right place by moving left or right
  (cond ((> (new 'value) (current 'value))
        (cond ((null? (current 'next)) ;; nothing else
              (connect current new))
            (else (ripple new (current 'next)))))
        ((null? (current 'previous))
         (connect new current))
        ((> (new 'value) ((current 'previous) 'value))
         ; insert between
         (connect (current 'previous) new)
         (connect new current))
        (else (ripple new (current 'previous)))))
```

**Question 7 (12 points):**

**Part a:**

```
(define factorials (cons-stream 1
  (mul-streams factorials integers)))
```

**Part b:** The definition of powers:

```
(define (powers x)
  (define pwr
    (cons-stream 1
      (scale-stream pwr x)))
  pwr)
```

**Part c:**

```
(define (exp-terms x)
  (div-streams (powers x) factorials))
```

**Part d:**

```
(define (exp-approx x)
  (define approx
    (cons-stream 0
      (div-streams (powers x) factorials))))
```

**Question 8 (10 points):**

**Part a:** A list of three pairs.

**Part b:** A list of three pairs, where the `car` of the second pair points to the third.

**Part c:** A list of three pairs, where the `car` of the first points to the second, and the `car` of the second points to the third.

**Part d:** Procedure would never return.

A list of three pairs, where the `car` of the last points to the first.

**Question 9 (23 points):**

**Part a:** Take the product of the even-valued leaves of the tree.

```
(tree-manip test-tree 1 (lambda (x) (if (even? x) x 1)) car cdr *)
```

--

**Part b:** Flatten a tree.

```
(tree-manip test-tree '() (lambda (x) (list x)) car cdr append)
```

**Part c:** Deep-reverse a tree.

```
(tree-manip test-tree '() (lambda (x) x) cdr car  
  (lambda (x y) (append x (list y))))
```

**Part d:** Create a new tree, which keeps the odd-valued leaves of the original tree within the same tree structure, but completely removes even-valued leaves.

```
(tree-manip test-tree '()  
  (lambda (x) (if (odd? x) x '()))  
  car  
  cdr  
  (lambda (x y) (if (null? x) y (cons x y))))
```