

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

Final Exam Solutions

Part 1. (36/200 points): Object oriented systems

Question 1:

```
> (ask eric 'WORK)
I've got this GREAT IDEA for a company!
please-call-me-PROFESSOR
  says: I need to PUBLISH my research but...
I love to hear myself TALK
```

Question 2:

```
> (ask eric 'PLAY)
Now that I am TENURED I can play all I want
```

Question 3:

```
get infinite loop of "I've got this GREAT IDEA for a company!"
```

Question 4:

```
> (ask eric 'WORK)
I've got this GREAT IDEA for a company!
I love to hear myself TALK
```

Question 5:

```
> (ask eric 'PLAY)
Now that I am TENURED I can play all I want
```

Question 6:

```
> (ask eric 'WORK)
please-call-me-PROFESSOR
  says: I need to PUBLISH my research but...
I love to hear myself TALK
```

Question 7:

```
> (ask eric 'PLAY)
Now that I am TENURED I can play all I want
```

Question 8:

```
> (ask eric 'WORK)
I love to hear myself TALK
```

Question 9:

```
> (ask eric 'PLAY)
ERIC
  says: I have NO TIME to play
I love to hear myself TALK
```

Part 2 (45/200 points): A Model of the Web

Procedures for the page and link abstractions are shown below. Note that the `data` argument for `make-page` has type `List<symbol or link>`.

```
(define (make-page url data) (list 'page url data))
(define (page? object) (and (pair? object) (eq? 'page (car object))))
(define (page-url page) (cadr page))
(define (page-data page) (caddr page))

(define (make-link word url) (list 'link word url))
(define (link? object) (and (pair? object) (eq? 'link (car object))))
(define (link-word link) (cadr link))
(define (link-url link) (caddr link))
```

Question 10:

Complete the following definition: `(define (replace! pred proc lst) ...)`

```
(if (null? lst)
    lst
    (begin (if (pred (car lst)) (set-car! lst (proc (car lst))))
           (replace! pred proc (cdr lst))))
```

Question 11:

Fill in the definition of `add-links!` below:

```
(define (add-links! page word url)
  (replace! ANSWER-A))

(lambda (x) (eq? x word))
(lambda (x) (make-link x url))
(page-data page)
```

Question 12:

Fill in the definition of `outgoing-urls` below, which uses `map` and `filter`.

```
(define (outgoing-urls page)
  (map ANSWER-B
       (filter ANSWER-C ANSWER-D)))
```

ANSWER-B:

`link-url`

ANSWER-C:

`link?`

ANSWER-D:

`(page-data page)`

Question 13:

Fill in the definition for `click` below.

```
(define (click page n)
  ANSWER-E)
```

ANSWER-E:

```
(let ((urls (outgoing-urls page)))
  (if (>= n (length urls))
      'not-found
      (list-ref urls n)))
```

Part 3 (20/200 points): Environment model**Question 14:**

```
(define foo
  (let ((old 0))
    (lambda (proc)
      (let ((return old))
        (set! old (proc old))
        return))))
```

Note that names are significant (foo, old, proc, return).

Question 15:

What expression involving an application of `foo` would lead to the second diagram?

```
(foo (lambda (x) (+ x 1)))
```

Question 16:

What expression involving an application of `foo` would lead to the third diagram:

```
(foo (lambda (y) (* y 3)))
```

Part 4 (50/200 points): Evaluators**Question 17:**

```
(define loop-clauses CADR)

(define loop-end CADDR)

(define loop-body CDDDR)

(define clause-var CAR)

(define clause-init CADR)

(define clause-update CADDR)

(define end-test CAR)

(define end-return CADR)
```

Question 18:

Here is a partially completed version of this procedure:

```
(define (eval-loop exp env)
  (let ((clauses (loop-clauses exp))
        (end (loop-end exp))
        (body (loop-body exp)))
    (let ((vars (map clause-var clauses))
          (inits (map clause-init clauses))
          (updates (map clause-update clauses)))
      (let ((new-env (extend-environment
                     ANSWER-A
                     ANSWER-B
                     ANSWER-C)))
        (loopit vars updates end body new-env))))))
```

ANSWER-A:

vars

ANSWER-B:

(map (lambda (val) (m-eval val env)) inits)

ANSWER-C:

env

Finally, you need to implement `loopit`

```
(define (loopit vars updates end body env)
  ANSWER-D)
```

ANSWER-D:

```
(if (true? (m-eval (end-test end) env))
    (m-eval (end-return end) env)
    (begin (eval-sequence body env)
            (let ((new-env (extend-environment
                            vars
                            (map (lambda (val) (m-eval val env)) updates)
                            env)))
              (loopit vars updates end body new-env))))))
```

Note a common mistake here will be interleaving evaluation of the update expressions with `set-variable-value!`. The semantics for loop say that all the update expressions must be evaluated *before* the variables are updated.

Question 19:

Provide code that completes the definition of `assert->if`.

```
(define (assert->if exp) ANSWER-E)
```

ANSWER-E:

```
(list 'if (assert-predicate exp)
      #t
      (list 'error
            "Assertion failed: "
            (list 'quote (assert-predicate exp))))
```

Solutions using back-quote and syntax procedures (like `make-if`) are also acceptable.

Part 5 (16/200 points): Streams

Question 20: Version 1; no memoization: E

Question 21: Version 1; with memoization: B

Question 22: Version 2; no memoization: A

Question 23: Version 2; with memoization: A

Part 6 (18/200 points): Orders of Growth**Question 24:**

```
(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst)))))
```

$T(n)$: **B** (linear) $S(n)$: **B** (linear) $H(n)$: **B** (linear)

Question 25:

```
(define (map2 proc lst)
  (define (helper rest answer)
    (if (null? rest)
        answer
        (helper (cdr rest)
                (append answer
                        (list (proc (car rest)))))))
  (helper lst '()))
```

where `append` is defined by

```
(define (append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1)
            (append (cdr lst1) lst2))))
```

$T(n)$: **D** (quadratic) $S(n)$: **B** (linear) $H(n)$: **D** (quadratic)

Question 26:

```
(define (f n)
  (cond ((= n 0) '())
        ((= n 1) (list 1))
        (else (list (f (- n 1)) (f (- n 2))))))
```

$T(n)$: **C** (exponential) $S(n)$: **B** (linear) $H(n)$: **C** (exponential)

Part 7 (15/200 points): General questions

Fill in the blank in each of the following statements.

Question 27:

A last-in, first-out (LIFO) data structure is called a **stack**.

Question 28:

Saying a language has **dynamic scoping** means that a procedure body is evaluated in an environment that extends the environment where the procedure was applied.

Question 29:

Saying a language has **lazy evaluation (or normal order evaluation)** means that operand expressions to a procedure are evaluated only when the operand's value is actually needed.

Question 30:

Languages with **tail recursion (or tail calls or tail-call optimization)** can implement iterative algorithms using recursive procedures.

Question 31:

Functional programming means programming without **side-effects (or mutation)**.

Question 32:

A **higher-order procedure** is defined as a procedure that takes a procedure as argument, returns a procedure as value, or both.