

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring 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. You may use scratch paper if you need, but the spaces we provide are the **only** places we will look at when grading.

Your name:

Circle Your Recitation Section		
Time	Instructor	T.A.
9	Una-May O'Reilly	Matt Debski
10	Una-May O'Reilly	Matt Debski/Shantanu Sinha
10	Saman Amarasinghe	Shantanu Sinha
11	Saman Amarasinghe	Mike Allen
10	Joe Stoy	Victor Hernandez
11	Joe Stoy	Victor Hernandez/Mike Allen
11	Mike Leventon	Sandia Ren
12	Mike Leventon	Hooman Vassef/Sandia Ren
12	Brian Scassellati	Eric Brittain
1	Brian Scassellati	Eric Brittain/Ron Weiss
1	Paul Penfield	Ron Weiss
2	Paul Penfield	Hooman Vassef

Please do not write below this line:

---

Problem	Value	Grade	Grader	Problem	Value	Grade	Grader
1	22			4	19		
2	20			5	15		
3	12			6	12		
				Total	100		

**Question 1 (22 points):**

Consider the following procedure:

```
(define memoize
  (lambda (proc)
    (let ((value '()))
      (lambda (arg)
        (let ((test (memq arg value)))
          (if test
              (cdar test)
              (let ((result (proc arg)))
                (set! value (cons (cons arg result)
                                   value))
                result))))))))
```

where

```
(define (memq key lst)
  (cond ((null? lst) '())
        ((= key (caar lst)) lst)
        (else (memq key (cdr lst)))))
```

Such a procedure can be used to create new procedures with memory, i.e. which keep track of previous evaluations and their results.

Suppose we evaluate the following:

```
(define square (lambda (x) (* x x)))

(define new-square (memoize square))

(new-square 6)
```

Attached to the end of the exam is a tear-off sheet, with some fragments from an environment diagram. You should EITHER complete this diagram directly on these fragments, OR you should do your own environment diagram on a separate sheet, then copy the labels for the fragments. In either case, answer the questions about the diagram on the following page.

First, for each procedure object, P1 through P6, identify, if possible, the environment pointer of the procedure, by circling the appropriate answer.

Procedure	Environment Pointer								
P1	E1	E2	E3	E4	E5	E6	GE	Not Shown	
P2	E1	E2	E3	E4	E5	E6	GE	Not Shown	
P3	E1	E2	E3	E4	E5	E6	GE	Not Shown	
P4	E1	E2	E3	E4	E5	E6	GE	Not Shown	
P5	E1	E2	E3	E4	E5	E6	GE	Not Shown	
P6	E1	E2	E3	E4	E5	E6	GE	Not Shown	

Second, for each environment frame, indicate which environment is the enclosing environment for that frame, by circling the appropriate answer.

Frame	Enclosing Environment									
E1	E1	E2	E3	E4	E5	E6	GE	None	Not Shown	
E2	E1	E2	E3	E4	E5	E6	GE	None	Not Shown	
E3	E1	E2	E3	E4	E5	E6	GE	None	Not Shown	
E4	E1	E2	E3	E4	E5	E6	GE	None	Not Shown	
E5	E1	E2	E3	E4	E5	E6	GE	None	Not Shown	
E6	E1	E2	E3	E4	E5	E6	GE	None	Not Shown	
GE	E1	E2	E3	E4	E5	E6	GE	None	Not Shown	

Finally, for each of the following variables, indicate to what it is bound in the diagram (possibilities include one of the procedure objects, P1, ..., P6, a number, a symbol, a list of numbers or symbols, an environment, E1, ..., E6, GE. Do this in terms of the values associated with these variables after **all** the expressions on page 2 have been evaluated.

Frame	Variable	Bound value
E1	x	
E2	result	
E3	arg	
E4	test	
E5	value	
E6	proc	
GE	memoize	
GE	square	
GE	new-square	

**Question 2 (20 points)**

We want to create a system for keeping an ordered sequence of elements, using message passing. Below is a procedure for creating elements of such sequences:

```
(define (make-unit)
  (let ((mine '()) ; value of unit
        (left '()) ; pointer to left neighbor
        (right '())) ; pointer to right neighbor
    (define me
      (lambda (msg)
        (cond
          ((eq? msg 'WHAT) <slot 1>)
          ((eq? msg 'SET!) <slot 2>)
          ((eq? msg 'LEFT) <slot 3>)
          ((eq? msg 'SET-LEFT!) <slot 4>)
          ((eq? msg 'RIGHT) <slot 5>))))
      me))
```

The idea is that each unit has a value that can be set, and has a left and right neighbor, which MUST be either empty or other “units”. Consider the following examples carefully – they illustrate the behavior and the values to be returned by different methods:

```
(define t1 (make-unit))
```

```
(define t2 (make-unit))
```

```
((t1 'SET!) 5)
;Value: done
```

```
(t1 'WHAT)
;Value: 5
```

```
((t2 'SET!) 3)
;Value: done
```

Consider the following example carefully – notice how setting a left neighbor also sets a right neighbor.

```
((t1 'SET-LEFT!) t2)
;Value: done
```

```
((t1 'LEFT) 'WHAT)
;Value: 3
```

```
((t2 'RIGHT) 'WHAT)
;Value: 5
```

Fill in the missing code fragments for the following:

<Slot 1>

<Slot 2>

<Slot 3> – assume slot 5 is symmetric

<Slot 4> – assume slot 6 is symmetric

**Question 3 (12 points):**

For each of the following expressions, which you are to treat as independent, a `cons` structure will result.

On the next page are a set of possible `cons` structures that may correspond to the result of each alternative.

Fill in the following table for each expression, identifying the resulting structure for `a`, or indicating if an error occurs in the evaluation, or stating if the structure is not shown in the set provided:

**A**

```
(define a (list (cons 1 2) 3 4))
(set-car! (cdr a) (car a))
```

**B**

```
(define a (list (cons 1 2) 3 4))
(set-cdr! a (car a))
```

**C**

```
(define a (list (cons 1 2) 3 4))
(set! a (car a))
```

**D**

```
(define a (list (cons 1 2) 3 4))
(set! (car a) (cdr a))
```

Circle the correct answer.

A	1	2	3	4	5	6	7	8	9	10	11	12	error	not shown
B	1	2	3	4	5	6	7	8	9	10	11	12	error	not shown
C	1	2	3	4	5	6	7	8	9	10	11	12	error	not shown
D	1	2	3	4	5	6	7	8	9	10	11	12	error	not shown

insert page of cons structures here

**Question 4 (19 points):**

One of the methods that we have examined for organizing systems of procedures involved using data directed programming. Suppose we want to create a system for simplifying algebraic expressions, using a data directed approach.

Below are some basic data abstractions for creating such expressions, which are exactly like the code shown in class:

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "bad tagged datum -- TYPE-TAG" datum)))

(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "bad tagged datum -- TYPE-TAG" datum)))

(define (constant? exp) (eq? (type-tag exp) 'number))

(define (make-constant x) (attach-tag 'number x))

(define (variable? exp) (eq? (type-tag exp) 'symbol))

(define (make-variable x) (attach-tag 'symbol x))
```

For example, we might create expressions such as:

```
(define one (make-constant 1))
(define two (make-constant 2))
(define x (make-variable 'x))
```

We want our system to simplify expressions, as follows:

```
(sum one two)
;Value: (number . 3)    NOTE -- SIMPLER NUMBER IN THIS CASE

(sum one x)
;Value: (sum (number . 1) (symbol . x))

(sum x one)
;Value: (sum (number . 1) (symbol . x))    NOTE ORDER OF RESULT

(sum (sum one two) x)
;Value: (sum (number . 3) (symbol . x))    NOTE -- SIMPLIFIES SUBEXPRESSIONS
```

Our means for doing this is to use the following interface:

```
(define (sum arg1 arg2)
  (apply-generic 'sum arg1 arg2))
```

where we have available a standard method for applying procedures, using a put-get table.

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (cons op args)))))) ; NOTE -- DIFFERENT FROM LECTURE VERSION
```

Remember that `put` installs a procedure into a table, indexed by two arguments, a symbol and a list of symbolic tags, and `get` retrieves the installed procedure using the same tags.

Write code to install a summing procedure for expressions with two numbers.

Write code to install a summing procedure for expressions with a symbol and a number.

**Question 5 (15 points):**

We want to create a procedure that when evaluated with a single argument, returns the value of the argument supplied the previous time it was evaluated. The desired behavior would be:

```
(test 1)
;Value: not-called-yet
```

```
(test 5)
;Value: 1
```

```
(test 8)
;Value: 5
```

Provide a definition of `test`.

**Question 6 (12 points):**

In problem set 5, you explored the use of a pattern matcher for simplifying expressions. The basic framework of the matcher was to use a set of rules, consisting of a pattern and a template. When the pattern was matched to an input expression, the template would be used to generate a set of new expressions, based on the match between the variables of the template and the input pattern. Remember that `?` variables match a single datum, while `~` variables match sequences of data.

Suppose we have a rule with pattern

`(?a ?b ?c)`

and template

`(?a * ?b * ?c)`

If this rule is matched to the expression

`(1 2 3)`

what are the possible new expressions created by instantiating the template?

Suppose we have a rule with pattern

`(~a ?b ~c)`

and template

`(~a * ?b * ~c)`

If this rule is matched to the expression

`(1 2 3)`

what are the possible new expressions created by instantiating the template?