

University of Massachusetts Lowell

---

91.301: Organization of Programming Languages  
Fall 2002

Quiz 1  
**Solutions to Sample Problems**

**Problem 1** What will Scheme print in response to the following statements? Assume that they are each evaluated in order in a single Scheme buffer. Write your answer below each statement. You may write “procedure” if a procedure [object] would be returned, or “error” if an error message would be returned. (This problem spans two pages.)

*To generate the solutions for this problem, we typed each expression to the Scheme interpreter (yes, yes, that’s cheating, but we wanted to make sure we had the Right Answer). Anywhere the interpreter returned a “[compound procedure ...]” expression, we accepted the answer “procedure,” and anywhere an error was triggered, we accepted “error.” The prefix “;Value: ” was not required in your answer.*

```
(define x 2)
```

```
;Value: "x --> 2" (read: "x is bound to 2"; the answer "x" was also accepted)
```

```
x
```

```
;Value: 2
```

```
(x)
```

```
;The object 2 is not applicable
```

```
;Type D to debug error, Q to quit back to REP loop:
```

```
(define (y) (* x 2))
```

```
;Value: "y --> #[compound-procedure 2 y]"
```

```
y
```

```
;Value: #[compound-procedure 2 y]
```

```
(y)
```

```
;Value: 4
```

```
(define (a) (lambda (x) (+ x 1)))  
;Value: "a --> #[compound-procedure 3 a]"
```

```
a  
;Value: #[compound-procedure 3 a]
```

```
(a)  
;Value: #[compound-procedure 4]
```

```
(let ((x 1)  
      (y 2)  
      (z (+ x 4)))  
  (+ x y z))  
;Value: 9
```

```
(define (if a b c) (+ a b c))  
;SYNTAX: define: redefinition of syntactic keyword if  
;Type D to debug error, Q to quit back to REP loop:
```

*Many students erroneously thought that if could be redefined (it cannot be redefined as it is a special form). This question was not penalized, as it is linked to the question below; meaning if you got this one wrong, you only lost points on the one below.*

```
(if 2 3 4)  
;Value: 3
```

**Problem 2** Write a function called `new-add` which returns the sum of two [ integers ]. Do not use the internal functions `+` and `-`, but instead, use `inc` (an already defined Scheme procedure which takes one argument and returns the sum of that argument and 1) and `dec` (a similar procedure which returns the sum of its argument and `-1`).

*All solutions to this problem require that an iteration variable count through one argument (without loss of generality, the first, `a`) and as that variable counts, the other argument (the second, `b`) is incremented or decremented and equal number of times.*

*The most elegant solution we could come up with (during discussions with students directly after the exam) is based on the observation that arranging the iteration variable to count from 0 up to `a`, whether `a` is positive or negative, allows us to use the same counting operation (`inc` or `dec`) on the iteration variable as will be used on the second argument `b`. We capture the counting operator `op` inside a `let`, and write a small helper function to iterate through the range `0-a`.*

```
(define (new-add a b)
  (let ((op (if (< 0 a)                ; determine the counting operator
              inc                       ; if going up, use increment
              dec)))                  ; if going down, use decrement
    (define (new-add-helper i sum)
      (if (= i a)                      ; done?
          sum                          ; yes, return the computed sum
          (new-add-helper (op i) (op sum)))) ; no, recurse (inc/dec i *and* sum)
    (new-add-helper 0 b)))            ; start running sum with b
```

*However, we accepted the following solution (and, like many of you, it was the first one we wrote):*

```
(define (new-add a b)
  (cond ((= a 0) b)                   ; return b if a has made it to zero
        (< a 0)                       ; counting up to 0 (is a negative)?
        (new-add (inc a) (dec b)))    ; yes, increment a, decrement b
        (else                          ; otherwise, counting down to 0
         (new-add (dec a) (inc b))))   ; so, decrement a, increment b
```

*The first solution is advantageous because only one comparison on `a` is ever made, and it might be considered more elegant all told. The second solution, despite the fact that `a` is examined for being above/below 0 on every iteration, is perhaps easier to understand.*

Is your procedure recursive?

yes

Is your procedure tail-recursive?

yes

Is the process it generates recursive or iterative?

iterative

**Problem 3** Assume that we have defined `sum` and `square` as follows:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
          (sum term (next a) next b))))

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

Determine the order of growth in time for the following functions using  $\Theta$  notation. (Hint: you will only need to use one or more of the following for your answers:  $\Theta(1)$ ,  $\Theta(\log n)$ ,  $\Theta(n)$ ,  $\Theta(n^2)$  and  $\Theta(2^n)$ . All classes might not be used.) Write your answer in the blanks to the right of each function.

```
(define (integrate a b f dx)
  (sum (lambda (x) (* (f x) dx))
      a
      (lambda (x) (+ x dx))
      b))
```

$\Theta(n)$

*The astute reader would realize that the answer here really depends on the order of  $f$ , which has not been specified (bonus points for those who wrote something about that). Our answer assumes that  $f$  is a constant-time function.*

```
(define (number-of-bits-in n)
  (if (< n 2)
      1
      (+ 1 (number-of-bits-in (/ n 2)))))
```

$\Theta(\log n)$

```
(define (times-5 x)
  (* x 5))
```

$\Theta(1)$

```
(define (exp a b)
  (cond ((< b 0) (error "Oops! b cannot be negative"))
        ((= b 0) 1)
        (else (if (odd? b)
                    (* a (exp a (- b 1)))
                    (square (exp a (/ b 2)))))))
```

$\Theta(\log n)$

```
(define (sum-of-squares x y)
  (+ (square x)
     (square y)))
```

$\Theta(1)$

```
(define (triangle-sum n)
  (sum (lambda (m) (sum (lambda (x) x) 1 inc m))
      1
      inc
      n))
```

$\Theta(n^2)$

**Problem 4** Write a procedure `power-close-to` that takes two non-zero positive integers (`b` and `n`) as arguments and returns the smallest power of `b` that is greater than `n`. That is, it should return the smallest integer  $i$  such that  $b^i > n$ . You may use the Scheme procedure `(expt b i)` which raises `b` to the power `i`.

*Our answer is a standard tail-recursive function which increments a counter `i` at each iteration, checking if the desired condition has been met. If so, the counter is returned as value (not the value `(expt b n)`); if not, the counter is incremented and recursive execution continues.*

*A fully-scored answer (with bonus) checked for the degenerate conditions when `b` is 1 ( $1^i = 1$  for all integers  $i > 0$ ), and the algorithm would otherwise enter an infinite loop.*

```
(define (power-close-to b n)
  (define (pct i)
    (if (> (expt b i) n)
        i
        (pct (inc i))))
  (if (= b 1)
      (error "Uh, sorry, there is no answer when b is 1!")
      (pct 1)))
```

*To assist in evaluating answers, the following helper procedure was written. This uses the function `format` which has not been discussed in class (but is described in the Scheme Info entry!) to format the output in a nice, readable way.*

```
(define (check-pct)
  (define (helper n)
    (if (< n 10)
        (let* ((b 2)
               (r (power-close-to b n)))
          (format #t
                  "~A^^A = ~@2A and should be greater than ~A~%"
                  b r (expt b r) n)
          (helper (inc n))))
    (newline)
    (helper 1)
    #t)
```

*Correctly running code should produce the following output (without checking on the degenerate case of `b = 1`).*

```
(check-pct)
2^1 = 2 and should be greater than 1
2^2 = 4 and should be greater than 2
2^2 = 4 and should be greater than 3
2^3 = 8 and should be greater than 4
2^3 = 8 and should be greater than 5
2^3 = 8 and should be greater than 6
2^3 = 8 and should be greater than 7
2^4 = 16 and should be greater than 8
2^4 = 16 and should be greater than 9
;Value: #t
```

Does your procedure generate an iterative process or a recursive process?

iterative

**Problem 5** A local bookstore has contracted aD University to provide an inventory system for their web site. We can create a database of books using Scheme. The constructor for a single book will be called `make-book` and takes the name of a book and its price as parameters.

```
(define (make-book name price)
  (cons name price))
```

Write the selectors `book-name` and `book-price`.

```
(define (book-name b) (car b))
(define (book-price b) (cdr b))
```

The inventory of books will be stored in a list. The selectors for our inventory data structure are `first-book` and `rest-books`, defined as follows:

```
(define first-book car)
(define rest-books cdr)
```

Write the constructor `make-inventory`.

*Any of*

```
(define make-inventory list)
(define (make-inventory . books) books)
(define make-inventory (lambda books books))
```

*would work.*

Draw the box-and-pointer diagram that results from the evaluation of

```
(define store-inventory
  (make-inventory (make-book 'sicp 60)
                 (make-book 'collecting-pep 15)
                 (make-book 'the-little-schemer 35)))
```

**Problem 5 (continued)** Write a procedure called `find-book` which takes the name of a book and an inventory as parameters and returns the book's data structure (name and price) if the book is in the store's inventory, and `[nil]` otherwise.

```
(define (find-book name inventory)
  (if ((null? inventory) nil)
      ((eqv? (book-name (first-book inventory)) name)
       (first-book inventory))
      (else
       (find-book name (rest-books inventory)))))
```

*Points were commonly lost here for breaking the abstraction barriers and for passing a book instead of the book's name.*

The bookstore has asked us to change our system to include a count of the number of copies of each book the store has on hand. We redefine our book constructor as follows:

```
(define (make-book name price num-in-stock)
  (list name price num-in-stock))
```

Write the selectors `book-name`, `book-price`, and `book-stock` for our new constructor.

```
(define book-name car)
(define book-price cadr)
(define book-stock caddr)
```

Will `find-book` need to be changed to accommodate our new representation? no (unless barriers were broken above)

**Problem 5 (continued)** Now that we are storing the number of copies in stock, write a procedure called `in-stock?` that takes a book name and an inventory as the parameters, and returns `#t` if at least one copy of the book is in stock, or `#f` otherwise. If the book is not listed in the inventory at all, `in-stock?` should also return `#f`. You may want to use your `find-book` procedure from above.

```
(define (in-stock? name inventory)
  (let ((book (find-book name inventory)))
    (and book
          (> 0 (book-stock book)))))
```

*Notice the use of `and` instead of `if`; remember that `and` evaluates its arguments in left-to-right order, each in turn, and only until the first evaluated argument is false (or it runs out of arguments). If all arguments evaluate non-false, then the value returned is the value of the last argument. If you used `if`, that would have been just fine.*

**Problem 6** Write a function `add-n` of one argument `n` that returns a procedure. The returned procedure takes one argument `x` and returns the sum of `x` and `n`.

```
(define (add-n n)
  (lambda (x) (+ x n)))
```

Using `add-n`, and without using the built-in Scheme procedure `*`, write `mult` which takes two integer arguments `a` and `b` and returns their product.

*There were three kinds of answers to this question. The first was along the lines of the formulation of `new-add` from the first quiz; the second used `repeated` (which worked only for positive values for the variable chosen for iteration, and thus lost points); the third was a very nice recursive formulation to deal with negative numbers. The three approaches are shown below.*

```
(define (mult a b)
  (let ((op-b (add-n b))           ; save the operator for b
        (op-i (if (< a 0) dec inc)) ; same for the iterator
        (define (m-helper i prod) ; the helper function to iterate
          (if (= i a)              ; are we done?
              prod                  ; yes, return the answer
              (m-help (op-i i) (op-b prod)))) ; no, advance counter and result
        (m-helper 0 0)))           ; start out at zero
```

```
(define (mult a b)                 ; works only for non-negative a
  ((repeated (add-n a) b) 0))      ; add b to 0 a times
```

```
(define (mult a b)
  (cond ((= a 0) 0)                ; done?
        ((< a 0)                   ; if negative a
         (- (mult (- a) b)))        ; then flip and continue
        ((add-n b) (mult (dec a) b))) ; iterate
```

**Problem 7** Assume the following expressions have been evaluated in the order they appear.

```
(define a (list (list 'q) 'r 's))
(define b (list (list 'q) 'r 's))
(define c a)
(define d (cons 'p a))
(define e (list 'p (list 'q) 'r 's))
```

Complete the table below with the result of applying the functions `eq?`, `eqv?`, and `equal?` to the two expressions on the left of each row. For example, the elements of the top row will represent the result from evaluating `(eq? a c)`, `(eqv? a c)`, and `(equal? a c)`. Your result should be written as `#t`, `#f` or *undefined* [or *unspecified*].

$\langle operand_1 \rangle$	$\langle operand_2 \rangle$	<code>eq?</code>	<code>eqv?</code>	<code>equal?</code>
a	c	<code>#t</code>	<code>#t</code>	<code>#t</code>
a	b	<code>#f</code>	<code>#f</code>	<code>#t</code>
a	(cdr d)	<code>#t</code>	<code>#t</code>	<code>#t</code>
d	e	<code>#f</code>	<code>#f</code>	<code>#t</code>
(car a)	(car e)	<code>#f</code>	<code>#f</code>	<code>#f</code>
(car a)	(cadr e)	<code>#f</code>	<code>#f</code>	<code>#t</code>
(caar a)	(caadr e)	<code>#t</code>	<code>#t</code>	<code>#t</code>

**Problem 8** Write `occurrences`, a procedure of two arguments `s` and `tree` that returns the number of times the first argument (an atom) appears in the second (a tree). You may find `accumulate-tree`, shown below, to be helpful.

```
(define (accumulate-tree tree term combiner null-value)
  (cond ((null? tree) null-value)
        ((not (pair? tree)) (term tree))
        (else (combiner (accumulate-tree (car tree) term combiner null-value)
                                         (accumulate-tree (cdr tree) term combiner null-value))))))
```

*Many, but not all of the students used `accumulate-tree` in their answer, as suggested by the hint.*

```
(define (occurrences s tree)
  (accumulate-tree tree
    (lambda (x) (if (eqv? x s) 1 0))
    +
    0))
```

*Those who did not use `accumulate-tree` ended up writing a procedure with the same functionality.*

**Problem 9** Draw the box and pointer diagrams for the following structures.

'(1)

'(1 2 3 4)

'((1))

'((1 . 2) (3 . 4))

'(1 (2 3) (4 (5) (6 7)))