

Structure and Interpretation of Computer Programs  
October 2000

Quiz 1  
**Scheme Basics**

Before starting, please write your name in the first blank below, and *optionally* a guess as to how well you think you will do in the second. Start the quiz only when instructed to do so. You may use any written resources you wish, but you may not consult another student, nor use a computer, nor a calculator. You will have one hour to finish this quiz, at which point please close the document, and *optionally* re-assess your anticipated grade in the third blank below. Please give your tests to the staff as you leave. Your actual grade will not be affected by your self-assessment, nor by opting out of self-assessment.

Name: \_\_\_\_\_ *Solutions* \_\_\_\_\_

Optional, expected grade (percent correct) before taking quiz: \_\_\_\_\_

Optional, expected grade (percent correct) after taking quiz: \_\_\_\_\_

P1: \_\_\_\_\_ P2: \_\_\_\_\_ P3: \_\_\_\_\_ P4: \_\_\_\_\_ Total: \_\_\_\_\_

**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 an 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