

This practice exam gives a sample of the type of questions that will appear on the actual final exam. All questions on the final will be multiple choice.

Answer the questions directly on the exam by circling the letter preceding the chosen answer. You may use the white space on the exam for your work. Correct answers receive full credit even with no work shown. Incorrect or alternate answers can sometimes receive full or partial credit based on work shown or comments given. Choose the BEST, SINGLE answer from the options provided. If you feel the need to make comments, then please use printing rather than cursive. Comments that are easily legible are more likely to be read.

The actual exam will have more questions than are given here. This practice is intended only as a sample of the type questions, topics and style of questions that will comprise the actual exam.

Students may use notes consisting of no more than one 8.5 x 11 inch sheet of paper. Students may use an English or English--other dictionary.

1) What is returned by the call to `cdr` below?

```
(define p '((a b c) x y z))  
(cdr p)
```

- a) (x y z)
- b) (a b c)
- c) ((a b c))
- d) (z)
- e) z

2) What is returned by the call to `cdr` below?

```
(define s '(a (d e f)))  
(cdr (cdr s))
```

- a) *cdr: expects argument of type <pair>; given ()*
- b) *cdr: expects argument of type <pair>; given (f)*
- c) (d e f)
- d) (e f)
- e) ()

3) Evaluate the scheme expression:

```
(+ 5 (/ (* (+ 10 5) 2 4) 5))
```

- a)  $17 \frac{1}{2}$
- b) 7.5
- c) 7.500000000000000000
- d) 29
- e)  $16 \frac{1}{5}$

4) What is

```
(eq? (null? '(a)) (null? '(b)))
```

- a) #f
- b) #t
- c) (#f)
- d) (#t)
- e) (#f #f)

5) What is

```
(bitwise-and 6 5)
```

- a) 6
- b) 5
- c) 4
- d) 1
- e) 0

6) What is

```
(bitwise-xor 6 5)
```

- a) 15
- b) 13
- c) 11
- d) 1
- e) 0

7) Using *g*, *h* and *k* as defined below, what is the value returned by `(k 5)`?

```
(define g
  (lambda (a b) (zero? (remainder a b))))

(define h
  (lambda (a b)
    (if (= a b) a
        (if (g a b) b (h a (+ b 1)))))

(define k (lambda (a) (h a 2)))
```

- a) 5
- b) 4
- c) 3
- d) 1
- e) 0

8) Using *g*, *h* and *k* as defined above, what is the value returned by `(k 91)`?

- a) 91
- b) 13
- c) 7
- d) 1
- e) 0

9) What is the output?

```
(let ((a 7) (b 13))
  (let ((b 3) (c (+ a b)))
    (let ((b 5))
      (cons a (cons b (cons c ())))))
  )
)
```

- a) (7 13 20)
- b) (7 5 20)
- c) (7 3 20)
- d) (7 (13 (20)))
- e) (7 (3 (20)))

10) What how could you make this code run much faster while maintaining its functionality?

```

1: (define roulette
2:   (lambda (myMoney)
3:     (define currentBet 1)
4:     (define maxBet 10000)
5:     (do ((i 0 (+ i 1))) ((>= i 100000000))
6:       (set! myMoney (- myMoney currentBet))
7:       (cond
8:         ((< (random) (/ 18 38))
9:          (set! myMoney
10:             (+ myMoney (* 2 currentBet)))
11:         (set! currentBet 1))
12:        (else
13:         (set! currentBet (* 2 currentBet))
14:         (if (> currentBet maxBet)
15:             (set! currentBet 1)
16:             )
17:         )
18:        )
19:       )
20:     myMoney
21:   )
22: )
23: (roulette 0)

```

- Move the definition of `currentBet` and `maxBet` outside of the definition `roulette`.
- Rewrite so that the values of `myMoney` and `currentBet` are updated using `(let ...)` rather than `(set! ...)`.
- Rewrite so that the values of `myMoney` and `currentBet` are updated using `(set ...)` rather than `(set! ...)`.
- Rewrite so that `roulette` takes two parameters: `currentBet`, and a counter that increments with each call. Then, replace the loop in line 5 with a recursive call to `roulette`.
- Change line 8 to: `((< (random) (/ 18.0 38.0))`

11) In a single run of the code above, how often is line 13 likely to be executed?

- about  $10,000 * 100,000,000$  times.
- about  $100,000,000$  times.
- about  $100,000,000 / 2$  times.
- about  $(18 / 38) * 100,000,000$  times.
- about  $(20 / 38) * 100,000,000$  times.

12) What is the output the following Scheme code?

```
(define foo
  (lambda x
    (apply + x)
  )
)

(foo 3 4 5)
```

- a) *procedure foo: expects 1 argument, given 3: 3 4 5*
- b) *procedure application: expected procedure, given: 3; arguments were: 4 5*
- c) *+: expects argument of type <number>; given (3 4 5)*
- d) 3
- e) 12

13) In mathematics, the *dot product*, also known both as the *scalar product* and the *inner-product*, is an operation which takes two vectors  $\vec{a} = (x_1, x_2, \dots, x_n)$ , and  $\vec{b} = (z_1, z_2, \dots, z_n)$  over the real numbers  $\mathfrak{R}$  and returns a real-valued

scalar quantity,  $s = \sum_{i=1}^n x_i z_i = x_1 z_1 + x_2 z_2 + \dots + x_n z_n$ .

In Scheme, if a and b are equal length lists of numbers, then the dot product of a and b is:

```
1: (define inner-product
2:   (lambda (a b)
3:     ?
4:   )
5: )
```

Where ? is replaced with:

- a) (map + (apply \* a b))
- b) (apply + (map \* a b))
- c) (map + (map \* a b))
- d) (apply + (apply \* a b))
- e) (map + (\* a b))

14) What is the output the following Scheme code?

```

1: (define f3 (lambda (myVector1)
2:   (define myVector2
3:     (apply vector (vector->list myVector1)))
4:   (vector-set! myVector2 0 (vector 77 88))
5:   (vector-set! (vector-ref myVector2 1) 0 333)
6:   (display myVector1)
7:   (display myVector2) (newline)
8: ))
9: (f3 (vector (vector 1 2) (vector 7 8)))

```

- a) `##(1 2) ##(333 8)##(77 88) ##(333 8)`
- b) `##(77 88) ##(333 8)##(77 88) ##(333 8)`
- c) `##(1 2) ##(7 8)##(77 88) ##(333 8)`
- d) `##(1 2) ##(7 8)##(1 2) ##(7 8)`
- e) `##(77 88) ##(7 8)##(77 88) ##(333 8)`

15) In lines 1 through 9 of the scheme code above, how many times is memory allocated for a new vector object?

- a) 3
- b) 4
- c) 6
- d) 7
- e) 8

16) What would be the output if line 4 were changed to

```
(vector-set! myVector2 0 77)?
```

- a) *vector-set!: index out of range for vector*
- b) *vector-set!: expects type <vector> as 1st argument*
- c) *vector-set!: expects type <vector> as 2st argument*
- d) *vector-ref: expects type <vector> as 1st argument*
- e) `##(1 2) ##(333 8)##(77 ##(333 8))`

17) What is the output of the Scheme code below?

- a) 0 0 0 0 2 3 2
- b) 0 1 2 3 4 6 5
- c) 0 1 2 4 6 3 5
- d) When subtree = node0, printTree gets stuck in an infinite loop.
- e) When subtree is any leaf node, printTree gets stuck in an infinite loop.

```
(define make-instance-node (lambda (myName)
  (define name myName)
  (define parent void)
  (define childList ())

  (define node (lambda argList
    (define cmd (car argList))
    (define args (cdr argList))
    (cond
      ((eq? cmd 'getName) name)
      ((eq? cmd 'setParent) (set! parent (car args)))
      ((eq? cmd 'getParent) parent)
      ((eq? cmd 'addChild)
       (set! childList (append childList args)))
      ((eq? cmd 'getChildren) childList)
    )
  ))
  node
))

(define addLeaf (lambda (nodeName parentNode)
  (define newNode (make-instance-node nodeName))
  (newNode 'setParent parentNode)
  (parentNode 'addChild newNode)
  newNode
))

(define printTree (lambda (subTree)
  (define childVector (list->vector (subTree 'getChildren)))
  (define n (vector-length childVector))
  (display (subTree 'getName))

  (do ((i 0 (+ i 1))) ((>= i n))
    (printTree (vector-ref childVector i))
  )
))

(define node0 (addLeaf "0 " void))
(define node1 (addLeaf "1 " node0))
(define node2 (addLeaf "2 " node0))
(define node3 (addLeaf "3 " node0))
(define node4 (addLeaf "4 " node2))
(define node5 (addLeaf "5 " node3))
(define node6 (addLeaf "6 " node2))

(printTree node0)
```