Programming Languages: Final Exam

Code: 201

1. An iterative process is generally characterized by executing in:

   (A) non-constant space
   (B) constant space
   (C) non-constant time
   (D) constant time

2. A Scheme function executes in constant space under what conditions? Choose the most general answer.

   (A) the number of non-tail recursive calls is zero
   (B) the number of tail recursive calls is zero
   (C) the number of non-tail recursive calls is unbounded
   (D) the number of non-tail recursive calls is bounded by a constant
   (E) the number of tail recursive calls is unbounded
   (F) the number of tail recursive calls is bounded by a constant

3. Consider the function:

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

   and its constant space rewrite:

```
(define (sum term a next b)
    (define (iter a result)
        (cond
            (UUU VVV)
            (else (iter WWW XXX))
            )
        )
    (iter YYY ZZZ)
    )
```

   What are valid expression for `XXX` and `ZZZ`, respectively? Assume a rewrite as advocated by the text.

   (A) `a` and `b`
   (B) none of the other answers are correct
   (C) `(next a)` and `0`
   (D) `b` and `(term a)`
   (E) `(+ a result)` and `b`
   (F) `(+ a (term a)` and `(next a)`
   (G) `(term a)` and `b`
   (H) `(+ (term a) result)` and `0`

4. Which functions implement iterative processes?

```
(define (sum n total)
    (cond
        ((= n 0)
            total)
        ((= (% n 2) 0)
            (+ (/ n 2) (sum (/ n 2) total)))
        (else
            (sum (- n 1) (+ total 1)))
        )
    )
```

```
(define (pow b e total)
    (cond
        ((= e 0)
            total)
         ((= (% e 2) 0)
            (pow (* b b) (/ e 2) total))
        (else
            (* b (pow b (- e 1) total)))
        )
    )
```

   (A) neither *sum* nor *pow*
   (B) *sum*
   (C) *pow*
   (D) *sum* and *pow*

5. What does this mystery function do?

```
(define mystery
    (lambda (p)
        (lambda (q)
            (lambda (r)
                (((lambda (f) (lambda (x) (f x))) q) ((p q) r))
                )
            )
        )
    )
```

(A) raises the Church numeral p to the q power
(B) increments the Church numeral p
(C) takes the base 1 log of the Church numeral p
(D) multiplies two Church numerals, p and q
(E) adds two Church numerals, p and q
(F) decrements the Church numeral p

6. **T** or **F**: Using the techniques from chapters 1 and 2 of the text (and those techniques **only**), it is possible to make a circular list of cons cells. In a circular list, the *cdr* of the last cons cell points to the first cons cell. A circular list of three cons cells, named *items*, with *car*s of 1, 2, and 3, would have the following behavior:

```
(car items) ; should be 1
(car (cdr items)) ; should be 2
(car (cdr (cdr items))) ; should be 3
(car (cdr (cdr (cdr items)))) ; should be 1
(car (cdr (cdr (cdr (cdr items))))) ; should be 2
(car (cdr (cdr (cdr (cdr (cdr items)))))) ; should be 3
(car (cdr (cdr (cdr (cdr (cdr (cdr items))))))) ; should be 1
...
```

7. Consider this version of *accumulate*:

```
(define (accumulate op base items)
    (cond
        ((null? items) base)
        (else (op (car items) (accumulate op base (cdr items))))
        )
    )
```

What kind of associativity does this function implement with regards to the operation *op*?

(A) high
(B) right
(C) low
(D) left

8. Continuing with the previous question, consider rewriting the accumulate function so that it implements the opposite associativity. Which of the following recursive calls is consistent with that goal? Hint: use a non-commutative operator to test the recursive call.

(A)
```
(op (cdr items)
  (accumulate op base (car items)))
```
(B)
```
(op (accumulate op base (car items))
  (cdr items))
```
(C)
```
(accumulate op base (op (car items)
  (accumulate op base (cdr items))))
```
(D)
```
(op (accumulate op base
    (cdr items)) (car items))
```
(E)
```
(accumulate op base
    (op (car items) (cdr items)))
```
(F)
```
(accumulate op base
    (cons (op (car items) (cadr items)) (cddr items))
```

9. Continuing with the previous question, how many base cases are absolutely necessary in the rewrite? Hint: test your rewrite with multiplication and a base of zero as well as accumulating an empty list.

(A) one, for the empty list
(B) four, for lists of length 0, 1, 2, and 3
(C) three, for lists of length 0, 1, and 2
(D) two, for a list of one item and for the empty list

10. With this implementation for *cons*:

```
(define (cons a b) (lambda (f) (if (f) a b)))
```

which of the following would be a valid function body for *cdr*, assuming the formal parameter is named *c*?

(A) `(c (if (f) a)`
(B) `(c (if (eq? f 'cdr) a b))`
(C) `(c #f)`
(D) `(c #t)`
(E) `(c (lambda () #f))`
(F) `(c (lambda () #t))`
(G) `(c (if (not (eq? f 'cdr)) a b))`
(H) `(c (if (not f) b))`

11. How many cons cells are produced when the expression:

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

is evaluated?

(A) 8
(B) 3
(C) 11
(D) 6
(E) 10
(F) 12
(G) 7
(H) 9

For the next set of questions, draw an environment picture in the style of the text after the following code is evaluated. Label the global environment *E0* and successive tables *E1*, *E2*, *E3*, ... as they come into existence.

12. Consider evaluating this code sequentially:

```
(define w 3)
(define (g x) (if (= x 0) 0 (g (- x 1))))
(define c (g w))
;mark
```

How many tables are created, *excluding* the global table, and how many of these point to the global table?

(A) three and one
(B) four and three
(C) four and four
(D) five and four
(E) two and one
(F) four and one
(G) three and three
(H) two and two

13. Continuing with the previous question, which tables may be garbage collected at the `;mark` comment?

(A) E2, E3, E4, E5
(B) E1
(C) E1, E2, E3, E4
(D) E1, E2, E3
(E) E2, E3, E4
(F) none of the other listed answers are correct
(G) E2, E3
(H) E1, E2

14. Assuming an empty global environment, consider the evaluation of this expression:

```
(define g (lambda (x) (lambda (y) (+ x y))))
```

How many closures are created?

(A) none of the listed answers are correct
(B) one under E0
(C) two under E1
(D) two under E0
(E) no closures are created
(F) one under E0, one under E1

15. Assuming an empty global environment, consider the sequential evaluation of these expressions:

```
(define g (lambda (x) (lambda (y) (+ x y))))
(define y (g 3))
(define x (y 5))
;mark
```

For each table, what is the count of objects (closures and other tables) that point to the table (at the `;mark` cbefore any garbage collection)? The counts are in parentheses immediately following the table label.

(A) E0 (2), E1 (1), E2 (0)
(B) E0 (2), E1 (1)
(C) E0 (1), E1 (2), E2 (1)
(D) E0 (2), E1 (1), E2 (1)
(E) none of the other answers are correct
(F) E0 (2), E1 (0)
(G) E0 (2), E1 (2)
(H) E0 (1), E1 (1)

16. What are the first five elements of the stream *s*:

```
(define s (cons-stream 1 (cons-stream 3 (add-streams s t))))
(define t (cons-stream 1 (cons-stream 3 (add-streams t (stream-cdr s)))))
```

(A) 1, 3, 4, 7, 10
(B) 1, 3, 4, 7, 11
(C) 1, 3, 4, 5, 10
(D) an error occurs
(E) 1, 3, 2, 6, 6
(F) 1, 3, 6, 11, 16
(G) 1, 3, 6, 6, 11
(H) an infinite loop occurs

17. What are the first eight values of stream *s*?

```
(define (add-shuffle-streams s t)
    (cons-stream
        (+ (stream-car s) (stream-car t))
        (add-shuffle-streams t (stream-cdr s))
        )
    )
(define s (cons-stream 0 (cons-stream 1 (add-shuffle-streams s (stream-cdr s)))))
```

Hint: `s[2]` is `s[0] + s[1]`, `s[3]` is `s[1] + s[1]`, `s[4]` is `s[1] + s[2]`, `s[5]` is `s[2] + s[2]`, and so on.

(A) none of the other answers are correct
(B) 0 1 1 2 2 2 3 4
(C) 0 1 1 2 2 3 3 5
(D) 0 1 2 3 5 8 13 21
(E) 0 1 1 2 3 5 8 13
(F) 0 1 0 1 0 1 0 1

18. What is wrong, if anything, with this implementation of *pairs*, which creates the first row of pairs and shuffles it with the subsequent rows of pairs?

```
(define (shuffle s t)
    (cons-stream (stream-car s) (shuffle t (stream-cdr s))))
(define (pairs s t)
    (shuffle
        (stream-map (lambda (x) (list (stream-car s) x)) t)
        (pairs (stream-cdr s) (stream-cdr t))
        )
    )
```

Given the streams `(0 1 2 3 4 5 ...)` and `(0 1 2 3 4 5 ...)`, the function *pairs* should produce the stream with the lists `(0 0)`, `(0 1)`, `(0 2)`, `(1 1)`, and so on, in some order.

(A) nothing
(B) the recursive call to *pairs* is not delayed
(C) the pairs with equal values (e.g. (1 . 1) and (2 . 2)) are omitted
(D) the pairs with equal values (e.g. (1 . 1) and (2 . 2)) are duplicated

19. A *separator* is a bit of punctuation that comes between two adjacent items in a sequence of items, whereas a *terminator* follows every item in a sequence of items. Which of the following sets of rules is consistent with a (possibly empty) sequence of items with terminator `XYZ`?

(i)
```
optSeq : *empty* | seq
seq : item | item XYZ seq
```

(ii)
```
optSeq : *empty* | seq
seq : item XYZ | item XYZ seq
```

(iii)
```
optSeq : *empty* | item | item XYZ optSeq
```

(A) (ii) and (iii)
(B) (iii)
(C) (i) and (iii)
(D) (ii)
(E) (i) and (ii)
(F) (i)

20. The following grammar rule is left recursive:

```
alpha : alpha DOG beta CAT alpha | RAT alpha
```

Which of the following sets of rules are semantically the same, but exhibit no left recursion:

(i)
```
alpha : RAT alpha gamma | *empty*
gamma : DOG beta CAT alpha
```

(ii)
```
alpha : RAT | gamma
gamma : DOG beta CAT alpha | *empty*
```

(iii)
```
alpha : RAT alpha gamma
gamma : DOG beta CAT alpha | *empty*
```

(A) (ii) and (iii)
(B) (i) and (ii)
(C) (iii)
(D) (ii)
(E) (i)
(F) (i) and (iii)