Programming Languages - Fall 2016

Exam 2

Code: 8373

1. What is the defining feature of function definitions in the lambda calculus? Choose the most complete answer:

   (A) one formal parameter, but multiple expressions in the body
   (B) multiple formal parameters, but one expression in the body
   (C) one formal parameter and one expression in the body
   (D) one formal parameter
   (E) multiple formal parameters and multiple expressions in the body
   (F) one expression in the body

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

3. Given the following definitions:

   ```
   ; Church numeral
   (define three (lambda (f) (lambda (x) (f (f (f x))))))
   ; incrementor
   (define (inc x) (list x '+ 1))
   ; original base value
   (define base 0)
   ```

   and this expression:

   ```
   (define x ((three inc) base))
   ```

   What is the value of x?

   (A) the list (3)
   (B) the list (0 + (1 + (1 + 1)))
   (C) the list (0 + 1 + 1 + 1)
   (D) the list ((((0) + 1) + 1) + 1)
   (E) the list (((0 + 1) + 1) + 1)
   (F) the list ((0 + (1 + (1 + 1))))
   (G) 3
   (H) the list (0 (+ 1 (+ 1 (+ 1))))

4. When Scheme prints a cons structure, it recursively prints the car and then recursively prints the cdr. However if the cdr is neither nil nor a cons cell, it prints a dot before printing the cdr. How many cons cells make up the structures with print-value of `(1 (2 . 3) 4)` and print-value `((1 . 2) (3 . 4))`, respectively?

   (A) 3 and 2
   (B) 4 and 5
   (C) 4 and 3
   (D) 4 and 4
   (E) 3 and 4
   (F) 3 and 3

5. In the language Schemey, the print-value of a cons structure uses a dot to indicate *every* cons cell. Consider this Schemey print-value:

   ```
   (1 . (2 . (3 . (4 . nil))))
   ```

   After drawing this structure, how would Scheme print this structure?

   (A) ((1 2 3) . 4)
   (B) ((((1) 2) 3) 4)
   (C) (((1 2) 3) 4)
   (D) (1 2 3 4 . nil)
   (E) (1 2 3 4)
   (F) (1 2 3 . 4)
   (G) (1 (2 (3 (4))))
   (H) (1 (2 (3 4)))

6. How many cons cells are needed, at a minimum, to represent an empty queue with a head and a tail pointer? Assume the variable q points to such a queue and that you must be able to access either the head or the tail pointer from q without using a conditional.

(A) 2
(B) 5
(C) 1
(D) 0
(E) 4
(F) 3

7. What is the length of the list produced by:

```
(map
    (lambda (x) (map (lambda (y) (list x y)) '(1 2 3)))
    '(8 7 6)
    )
```

(A) the result is not a list
(B) 3
(C) 1
(D) 6
(E) 18
(F) 2
(G) 9
(H) 27

8. Suppose you wish to flatten a list m that contains lists of atoms (an atom is anything that is not a cons cell).

```
(flatten '((a b) (c d) (e))) ; should evaluate to (a b c d e)
```

Which of the following expressions accomplishes that task? Choose the most simple answer. These helper functions might be useful:

```
(define h1 (lambda (x) (if (pair? x) cons append)))
(define h2 (lambda (a b) ((if (pair? a) cons append) a b)))
```

(A) `(accumulate h2 nil m)`
(B) `(map append m)`
(C) `(map h2 m)`
(D) `(accumulate append nil m)`
(E) `(map h1 m)`
(F) `(accumulate h1 nil m)`

9. Consider a function called *selective-map* that applies a function to each of the items in a given list *if* a predicate function returns true for that item. If the predicate function returns false, that element appears in the new list unchanged. Here is the function signature:

```
(define (selective-map p? f items) ...)
```

where *p?* is the predicate and *f* is the function to be applied. What is a valid implementation of the body of *selective-map*? Assume the *filter* function keeps those elements for which the given predicate is true.

(A) (map f (filter p? items))
(B) (filter p? (map f items))
(C) none of the other answers are correct
(D) (map p? (filter f items))
(E) (filter f (map p? items))

10. Suppose you wish to make a version of *selective-map* named *selective-map2* that applies the given function to an element if the predicate returns false. Assuming a similar function signature to that of *selective-map*, what is a valid implementation of this new function?

(A) `(selective-map (not p?) f items)`
(B) none of the other answers are correct
(C) `(not (selective-map p? f items))`
(D) `(selective-map (lambda (x) (not (p? x))) f items)`
(E) `(selective-map (lambda (x) ((not p?) x)) f items)`

11. Which of the following expressions is equivalent to `(map f m)`?

(A) `(accumulate (lambda (x y) (cons x (f y))) nil m)`
(B) `(accumulate (lambda (x y) (cons (f x) y)) nil m)`
(C) `(accumulate (lambda (x) (f x)) nil m)`
(D) none of the listed expressions work
(E) `(accumulate (lambda (x) (cons (f x) m)) nil m)`

12. Suppose you wish to use *accumulate* to count the number of items in a list *m*. Which of the following expressions accomplishes that task?

(A) none of these expressions work
(B) `(accumulate (lambda (x) (+ 1 (cdr m))) 0 m)`
(C) `(accumulate (lambda (x y) (+ 1 (cdr y))) 0 m)`
(D) `(accumulate (lambda (x) 1) 0 m)`
(E) `(accumulate (lambda (x y) (+ 1 y)) 0 m)`

13. Suppose you wish to use *map* to count the number of items in a list *m*. Which of the following expressions accomplishes that task?

(A) `(map (lambda (x y) (+ 1 y)) m)`
(B) `(map (lambda (x y) (+ 1 (cdr y))) m)`
(C) `(map 1 m)`
(D) none of these expressions work
(E) `(map (lambda (x) 1) m)`
(F) `(map (lambda (x) (+ 1 (apply + (cdr m)))) m)`

14. Suppose you are tasked to count the number of items in a cons structure that are *not* cons cells. What is the minimum number of cases that need to be processed in your *cond*, including any base cases? You may not use *and* or *or* in your *cond* test cases. Assume the existence of the *pair?* predicate, which returns true if its given argument is a cons cell. Here are some examples:

```
(countItems nil)                  ; should evaluate to 0
(countItems 5)                    ; should evaluate to 1
(countItems (list 1 2 3))         ; should evaluate to 3
(countItems (list (cons 1 2) 3 4)) ; should evaluate to 4
```

(A) 5
(B) 2
(C) 3
(D) 4

15. In the symbolic differentiation example in the text, the only legal entities are numbers and tagged lists. Suppose we wish to differentiate an exponentiation and simplify the result. What is the minimum number of cases in a *cond* where a result could be simplified in a *make-exponentiation* function? You may not use *and* or *or* in your *cond* cases. Recall that 1^n, 1^0, n^0, and 0^0 all evaluate to 1, while n^1 is n and 0^n is 0. Assume all bases and exponents are non-zero.

(A) 5
(B) 1
(C) 6
(D) 2
(E) 3
(F) 4

16. Consider making *make-multiplication* in the differentiation system variadic, allowing two or more operands:

```
(define (make-multiplication @) (cons '* @))
```

What would be a valid implementation of the selector *multiplier*, which should return the right-hand operand, for the case when there are three or more operands?

(A) `(apply make-multiplication (caddr product))`
(B) `(make-multiplication (cddr product))`
(C) `(make-multiplication (cdddr product))`
(D) `(apply make-multiplication (cdddr product))`
(E) `(apply make-multiplication (cadddr product))`
(F) `(make-multiplication (caddr product))`
(G) `(apply make-multiplication (cddr product))`
(H) `(make-multiplication (cadddr product))`

17. What is the print value of the expression `'(quote ''a)`?

(A) `(quote (quote (quote quote a)))`
(B) `(quote quote quote a)`
(C) `(quote (quote (quote (quote a))))`
(D) `(quote (quote (quote a)))`
(E) `(quote (quote a))`
(F) none, it's an invalid expression
(G) `(quote quote quote quote a)`
(H) `(quote quote (quote quote a))`

18. Consider implementing the *equal?* predicate, intended to work on numbers, symbols, and cons structures. is a partially finished implementation:

```
(define (equal? a b)
    (cond
        ((null? b) (null? a))
        ((and (pair? a) (pair? b)) YYY)
        (else XXX)
        )
    )
```

where `XXX` and `YYY` represent expressions to be completed. What are valid expressions that can be substituted for `XXX` and `YYY`, respectively. Note: the *eq?* predicate can be used for pointer, symbolic, and numeric equality.

(A) `(equal? a b)` and `(and (eq? (car a) (car b)) (eq? (cdr a) (cdr b)))`

(B) `(equal? a b)` and `(and (eq? (car a) (car b)) (equal? (cdr a) (cdr b)))`

(C) `(equal? a b)` and `(and (equal? (car a) (car b)) (equal? (cdr a) (cdr b)))`

(D) `(eq? a b)` and `(and (eq? (car a) (car b)) (equal? (cdr a) (cdr b)))`

(E) `(eq? a b)` and `(and (eq? (car a) (car b)) (eq? (cdr a) (cdr b)))`

(F) `(eq? a b)` and `(and (equal? (car a) (car b)) (equal? (cdr a) (cdr b)))`

19. Consider re-implementing *apply-generic* (from the text) for unary operations on tagged data. Here is the function signature:

```
(define (apply-generic op item) ...)
```

Assuming *get* and *tupe-tag* have their definitions given in the text, what would *not* be a valid implementation of the body? Note that the actual function stored in the table is responsible for unpacking any tagged lists. Choose the correct answer with the fewest number of printable characters.

(A) `(apply (get op (list (type-tag item))) (list item))`

(B) `((get op (map type-tag (list item))) item)`

(C) `(apply (get op (map type-tag (list item))) (list item))`

(D) `(apply (get op (type-tag item)) item)`

(E) `((get op (list (type-tag item))) item)`

20. What is the minimum number of cons cells needed to implement an environment table that contains a two variables bound to integers. Assume the variables and their values are kept as parallel lists. Assume the table points to the global environment.

(A) none of the listed counts are correct

(B) 6

(C) 4

(D) 8

(E) 2

(F) 5

(G) 7

(H) 3

21. **T** or **F**: One can create an empty global environment with the function used for extending an existing environment by passing in null pointers for the variables list, the values list, and the existing environment.

22. **T** or **F**: Under modern static scoping rules, there is no need to extend an environment if the function being called has no formal parameters.