Programming Languages

Exam 1

Fall 2016

Code: 3165

Write all code in Scheme/Scam unless otherwise directed. No form of assignment is allowed.

1. Consider writing a iterative process function to compute the number of mod operations that *gcd*:

```
(define (gcd m n)
    (if (= n 0)
        m
        (gcd n (% m n))
        )
    )
```

would perform under normal evaluation with a priori knowledge of how many recursive calls are made:

```
(define (countMods recursiveCallCount)
    (define (iter x y n)
        (cond
            ((= n 0) ???)
            (else (iter ??? ??? (- n 1)))
            )
        )
    (iter 0 0 recursiveCallCount)
    )
```

This function is intended to simulate the sequence:

```
(gcd m n)
(gcd n (% m n))
(gcd (% m n) (% n (% m n)))
...
```

Assuming that x tracks the number of mod operation found in the first argument to gcd and y tracks the number in the second, what are the updates to x and y, respectively?

(A) `y` and `(+ x y)`
(B) `(+ x 1)` and `(+ y 1)`
(C) `y` and `(* y 2)`
(D) `y` and `(+ x y 1)`
(E) `(+ x 1)` and `(* y 2)`
(F) `(+ x 1)` and `(+ (* y 2) x)`

2. Continuing with the previous question, what is the base case return value in helper function?

(A) `(* x y))`
(B) `y`
(C) `(+ x y 1)`
(D) `(+ (* x y) 1)`
(E) `x`
(F) `(+ x y)`

3. **T** or **F**: It is possible to program a loop in a purely *functional* programming language.

4. **T** or **F**: A purely *functional* programming language cannot loop in constant space.

5. **T** or **F**: A purely *functional* programming language cannot have assignment (i.e. a variable can change its value during its lifetime).

6. **T** or **F**: Scheme's variadic mathematical operators (e.g. `+`, `-`) have precedence (e.g. multiplications are performed before additions).

7. **T** or **F**: Scheme's variadic mathematical operators (e.g. `+`, `-`) have associativity (i.e. arguments are preferentially combined by dictate of the language specification).

8. **T** or **F**: An iterative process whose iteration count depends on the size of the input must have the same time and space complexity.

9. What kind of recursion does the following function exhibit and what kind of process does it implement?

```
(define (g a b c)
    (cond
        ((= a 1) (* b c))
        ((= (% a 2) 0) (g (/ a 2) (* b 2) c))
        (else (g (- a 1) b (+ c a)))
        )
    )
```

(A) it's syntactically non-tail recursive, so iterative
(B) it's syntactically non-tail recursive, so recursive
(C) it's syntactically tail recursive, so recursive
(D) it's syntactically tail and non-tail recursive, but overall iterative
(E) it's syntactically tail recursive, so iterative
(F) it's syntactically tail and non-tail recursive, but overall recursive

10. What kind of recursion does the following function exhibit and what kind of process does it implement?

```
(define (g n m)
    (cond
        ((< n 2) (* n m))
        ((> m 0) (g (- n 1) (+ m 1)))
        ((= m 0) (+ 1 (g (/ n 2) 1)))
        )
    )
```

(A) it's syntactically tail and non-tail recursive, but overall iterative
(B) it's syntactically non-tail recursive, so iterative
(C) it's syntactically tail recursive, so iterative
(D) it's syntactically tail and non-tail recursive, but overall recursive
(E) it's syntactically non-tail recursive, so recursive
(F) it's syntactically tail recursive, so recursive

11. What kind of recursion does the following function exhibit and what kind of process does it implement?

```
(define (f a r)
    (cond
        ((= a 1) r)
        ((< a 8) (+ 1 (f (/ a 2) r)))
        (else (f (/ a 2) (+ r 1)))
        )
    )
```

(A) it's syntactically tail and non-tail recursive, but overall recursive
(B) it's syntactically non-tail recursive, so iterative
(C) it's syntactically tail recursive, so recursive
(D) it's syntactically tail recursive, so iterative
(E) it's syntactically tail and non-tail recursive, but overall iterative
(F) it's syntactically non-tail recursive, so recursive

12. Suppose *g* is defined as:

```
(define (g f)
    (lambda ()
        (if (integer? f) (+ f f) (f 4))
        )
    )
```

To what does the expression `((g g))` evaluate?

(A) 8
(B) 4
(C) an anonymous function with no formal parameters
(D) an error (int called as a function)
(E) the closure bound to *g*
(F) the composition of *g* and *g*

13. Given function *g* as defined above, to what does `(((g g)))` evaluate?

   (A) the composition of *g* and *g*
   (B) 4
   (C) an anonymous function with no formal parameters
   (D) 8
   (E) the closure bound to *g*
   (F) an error (int called as a function)

14. Consider this accumulate function:

```
(define (accumulate op base lo hi)
    (cond
        ((= lo hi) base)
        (else (op lo (accumulate op base (+ lo 1) hi)))
        )
    )
```

   What associativity does this function implement, in terms of combining the numbers from *lo* to *hi*?

   (A) left
   (B) there is no associativity with prefix notation
   (C) it depends on the operator
   (D) right

15. Consider the function signature:

```
(define (function+ g f) ...)
```

   The intent of *function+* is to compose two functions *g* and *f* together. For example, the call

```
((function+ cube sqrt) 4)
```

   would evaluate to 8, since the square root of 4 is 2 and the cube of 2 is 8. What is a valid body for this composing function?

   (A) `((lambda (x) (f (g x))) 4)`
   (B) `(lambda (x) (f x) (g x))`
   (C) `((lambda (x) (g (f x))) 4)`
   (D) `(lambda (x) (g x) (f x))`
   (E) `(lambda (x) (f (g x)))`
   (F) `((lambda (x) (g x) (f x)) 4)`
   (G) `(lambda (x) (g (f x)))`
   (H) `((lambda (x) (f x) (g x)) 4)`

16. Consider rewriting the following Scam function, removing the local defines by converting the body of the function to a lambda definition and subsequent call to that lambda:

```
(define (f x)
    (define a (+ x 1))
    (define b (* a a)) ; note that b depends on the a defined on the previous line
    (define (g c) (* c c))
    (+ (* a a) (* (g a) a) (* (g b) b)) ; action
    )
```

   Note that in Scam, unlike Scheme, local definitions are processed sequentially. Thus a subsequent local definition can refer to a previous one.

   The rewrite must be semantically equivalent to the old definition and the action of the function (i.e. the line of code so marked) must not change. How many lambda definitions wrap the action in a *minimal* rewrite?

   (A) only one lambda, no nesting
   (B) two lambdas, placed sequentially
   (C) the rewrite cannot be performed in Scam, only in Scheme
   (D) two lambdas, one nested inside the other

17. Consider rewriting the following Scheme function, removing the local defines by converting the body of the function to a lambda definition and subsequent call to that lambda:

```
(define (f x)
    (define a (+ x 1))
    (define b (* a a)) ; note that b depends on a non-local a
    (define (g c) (* c c))
    (+ (* a a) (* (g a) a) (* (g b) b)) ; action
    )
```

Note that in Scheme, unlike Scam, local definitions may be processed in any order. Thus a subsequent local definition cannot refer to a previous one.

The rewrite must be semantically equivalent to the old definition and the action of the function (i.e. the line of code so marked) must not change. How many lambda definitions wrap the action in a *minimal* rewrite?

(A) two lambdas, one nested inside the other
(B) two lambdas, placed sequentially
(C) only one lambda, no nesting
(D) the rewrite cannot be performed in Scheme, only in Scam

18. This function:

```
(define (f m n)
    (cond
        ((= n 0) 1)
        ((= (% n 2) 0) (f (* m m) (/ n 2)))
        (else (* m (f m (- n 1))))
        )
    )
```

executes in:

(A) linear time and constant space
(B) log time and constant space
(C) log time and linear space
(D) linear time and log space
(E) linear time and linear space
(F) log time and log space

19. This function:

```
(define (f m n)
    (cond
        ((= n 0) 1)
        ((= (% n 2) 0) (square (f m (/ n 2))))
        (else (* m (f m (- n 1))))
        )
    )
```

executes in:

(A) linear time and linear space
(B) linear time and constant space
(C) log time and linear space
(D) log time and constant space
(E) linear time and log space
(F) log time and log space

20. What is this version of Fermat's little theorem in Scheme notation:

> Given a prime number n and an integer a less than n (both positive), then a raised to the n^{th} power is congruent to a, modulo n.

(A) `(= (% (^ a n) n) a)`
(B) `(= (% (^ a n) n) (% n a))`
(C) `(= (^ a (% n a)) (% a n))`
(D) `(= (% (^ a n) a) (% a n))`

21. What is the drawback of the following implementation with respect to primality testing:

```
(define (expmod base exp m)
    (cond
        ((= exp 0) 1)
        ((even? exp) (% (* (expmod base (/ exp 2) m) (expmod base (/ exp 2) m)) m))
        (else (% (* base (expmod base (- exp 1) m)) m))
        )
    )
```

(A) *expmod* may overflow in systems with fixed-sized integers
(B) *expmod* runs in exponential time
(C) *expmod* performs redundant calculations
(D) the mod operation should happen before the call the recursive call
(E) *expmod* is incorrect for exponents of one and zero

22. For the problem above, what recurrence describes the even case?

(A) T(n) = 2T(n/2) + O(log n)
(B) T(n) = T(n-1) + O(log n)
(C) T(n) = 2T(n/2) + O(1)
(D) T(n) = T(n/2) + O(log n)
(E) T(n) = T(n/2) + O(1)
(F) T(n) = 2T(n-1) + O(log n)
(G) T(n) = T(n-1) + O(1)
(H) T(n) = 2T(n-1) + O(1)

23. Suppose you wish to pass function *f* to function *g*. The correct syntax would be:

(A) `g((f))`
(B) `((g (f)))`
(C) `g(f)`
(D) `(g (f))`
(E) `(g f)`
(F) `(g)(f)`

24. To what does the following expression evaluate?

```
((lambda (x) ((lambda (x) (+ x x)) 6)) 10)
```

(A) 12
(B) an error (nested lambdas with the same formal parameters)
(C) an error (int called as a function)
(D) a function with formal parameter x
(E) 20
(F) 16

25. To what does the following expression evaluate?

```
((lambda (x) ((lambda (x y) (- x y)) 10 6)))
```

(A) an error (too few arguments)
(B) a function with formal parameter x
(C) a function with formal parameter y
(D) 4
(E) -4
(F) an error (int called as a function)

26. To what does the following expression evaluate?

```
((lambda (x) (lambda (y) (- x y))) 10)
```

(A) an error (int called as a function)
(B) -4
(C) a function with formal parameter y
(D) a function with formal parameter x
(E) 4

27. Consider transforming the following function to one that iterates in constant space, as described in class:

```
(define (f x y z)
    (cond
        ((< z 2) 1)
        ((= z 2) (* x y z))
        ((= (% z 2) 0) (+ (f x y (- z 3)) (f x y (- z 2))))
        (else (* y (f (- x 1) y (- z 1))))
        )
    )
```

Assuming the above function is only valid for non-negative numbers, how many formal parameters will the constant space iterating function have?

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

28. Consider transforming the following grammar rule into a recursive descent recognizer rule, as described in class:

```
alpha : beta gamma
      | AYE BEE delta
      | beta delta CEE
      | AYE AYE
      | CEE BEE
```

Assuming that any default (else) case assumes the remaining alternative and that a call to *check* is preferred over a call to a pending function, how many calls to check and how many calls to pending functions are made, respectively, in a minimal implementation?

(A) 4 and 1
(B) 6 and 4
(C) 3 and 3
(D) 4 and 2
(E) 7 and 4
(F) 4 and 3
(G) 3 and 1
(H) 3 and 2

29. Given the grammar rule above, the resulting function features how many calls to *match*, again assuming a minimal implementation. Assume also that *match* is the only way to advance the lexical stream.

(A) 3
(B) 7
(C) 0
(D) 8
(E) 6
(F) 5

30. Given the grammar rule above, the associated pending function features how many calls to *match*, calls to *check*, and calls to other pending functions, respectively?

(A) 0, 2, and 2
(B) 0, 1, and 1
(C) 0, 3, and 2
(D) 2, 2, and 2
(E) 1, 2, and 1
(F) 0, 2, and 1
(G) 2, 3, and 2
(H) 1, 1, and 1