Programming Languages
Exam 1
Spring 2016

Code: 48660

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

Multiple choice and True/False questions are worth one point each.

1. **T** or **F**: A purely *functional* programming language cannot have loops.
2. **T** or **F**: A purely *functional* programming language cannot have loops that execute in constant space.
3. **T** or **F**: A purely *functional* programming language cannot have assignment (i.e. a variable can change its value during its lifetime).
4. **T** or **F**: In Scheme, `+` is an operator, not a function.
5. **T** or **F**: In Scheme, `+` can be redefined.
6. **T** or **F**: In most imperative languages (e.g. C, C++, Java), keywords cannot be variables.
7. **T** or **F**: Scheme's variadic mathematical operators (e.g. `+`, `-`) have precedence (e.g. multiplications are performed before additions).
8. **T** or **F**: Scheme's variadic mathematical operators (e.g. `+`, `-`) have associativity (i.e. arguments are preferentially combined by dictate of the language specification).
9. **T** or **F**: If a language implements any kind of precedence, the language processor must perform multiplications before additions when given a choice.
10. **T** or **F**: An iterative process whose iteration count depends on the size of the input executes in constant space.
11. **T** or **F**: A recursive process whose iteration count depends on the size of the input cannot execute in constant space.
12. **T** or **F**: A recursive function cannot implement an iterative process.
13. **T** or **F**: A recursive process has the same time and space complexity (e.g. if it executes in linear space, it must execute in linear time, and vice versa)
14. **T** or **F**: An iterative process whose iteration count depends on the size of the input must not have the same time and space complexity.
15. What kind of process does the following function implement?

```
(define (f c b a)
    (cond
        ((= a 0) (* b c))
        ((< a 5) (f b c (+ a 7)))
        ((< a 10) (+ b (* 2 c) (f (- a 1) b c)))
        (else (+ a b c))
        )
    )
```

(A) it's syntactically non-tail recursive, so recursive

(B) it's syntactically tail and non-tail recursive, but overall iterative

(C) it's syntactically tail recursive, so iterative

(D) it's syntactically tail and non-tail recursive, but overall recursive

16. What kind of process does the following function implement?

```
(define (f b a)
    (cond
        ((= b 0) a)
        ((< b 3) (f (- b 1) (* a b)))
        (else (* a (f (- b 1) (+ a 1))))
        )
    )
```

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

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

```
(define (f n)
    (f (+ n 1))
    0
    )
```

(A) non-tail and recursive
(B) it's an infinite loop, so those designations don't apply
(C) non-tail and iterative
(D) both tail and non-tail and recursive
(E) tail and iterative
(F) tail and recursive
(G) both tail and non-tail and iterative

18. 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) tail and iterative
(B) non-tail and recursive
(C) both tail and non-tail and iterative
(D) non-tail and iterative
(E) both tail and non-tail and recursive
(F) tail and recursive

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

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

(A) non-tail and recursive
(B) tail and recursive
(C) tail and iterative
(D) non-tail and iterative
(E) both tail and non-tail and recursive
(F) both tail and non-tail and iterative

20. Suppose g is defined as:

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

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

(A) g
(B) f
(C) 8
(D) the composition of g and g
(E) 2
(F) 4

21. Consider this accumulate function:

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

To what would the call `(accumulate + 1 1 5)` evaluate?

(A) 1
(B) 11
(C) 0
(D) 10
(E) the code generates an error

22. Continuing with the previous *accumulate* function, to what would the call `(accumulate * 0 2 5)` evaluate?

(A) the code generates an error
(B) 1
(C) 24
(D) 0
(E) 25

23. Continuing with the previous *accumulate* function, what kind of associativity is implemented (or intended)?

(A) right
(B) none
(C) left
(D) middle

24. Consider this function:

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

To what would the call `(accumulate - 0 1 5)` evaluate?

(A) -11
(B) the code generates an error
(C) -8
(D) -1
(E) -10

25. Continuing with the previous *accumulate* function, to what would the call `(accumulate * 1 2 5)` evaluate?

(A) 0
(B) 25
(C) 24
(D) 1
(E) the code generates an error

26. Continuing with the previous *accumulate* function, what kind of associativity is implemented (or intended)?

(A) right
(B) left
(C) none
(D) middle

27. Consider the function signature:

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

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

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

would evaluate to 8, since the cube of 4 is 64 and the square root of 64 is 8. What is a valid body for this composing function?

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

28. Section 1.3.2 in the text discusses how to use lambdas in place of locally defined variables. Consider doing a straightforward rewrite of this function, replacing its local definitions in the prescribed manner:

```
(define (f x)
    (define a (+ x 1))
    (define b (- x 1))
    (define (g c) (* c c))
    (+ (* a a) (* (g a) a) (* (g b) b)) ; action
    )
```

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

(A) 4
(B) 2
(C) 3
(D) 1

29. Continuing with the previous question, what are valid formal parameters lists for the lambda(s) that wrap the action in a minimal rewrite?

(A) `(lambda (a) ...)` and `(lambda (b) ...)` and `(lambda (g) ...)`
(B) `(lambda (a b) ...)` and `(lambda (g) ...)`
(C) `(lambda (a b g) ...)` and `(lambda (c) ...)`
(D) `(lambda (a b) ...)` and `(lambda (g c) ...)`
(E) `(lambda (x) ...)`
(F) `(lambda (a b g) ...)`
(G) `(lambda (a) ...)` and `(lambda (b) ...)` and `(lambda (g) ...)` and `(lambda (c) ...)`

30. Continuing with the previous question, what are valid arguments for the lambda(s) that wrap the action in a minimal rewrite?

(A) `((lambda ...) (+ a 1) (- b 1))` and `((lambda ...) (lambda (c) (* c c)))`
(B) `((lambda ...) (+ x 1) (- x 1))` and `((lambda ...) g (* c c))`
(C) `((lambda ...) (+ a 1) (- b 1) (lambda (c) (* c c)))`
(D) `((lambda ...) x)`
(E) `((lambda ...) (+ x 1) (- x 1) (lambda (c) (* c c)))`
(F) `((lambda ...) (+ x 1) (- x 1) g)` and `((lambda ...) (* c c))`
(G) `((lambda ...) (+ x 1)` and `((lambda ...) (- x 1))` and `((lambda ...) g)` and `((lambda ...) c)`
(H) `((lambda ...) (+ x 1)` and `((lambda ...) (- x 1))` and `((lambda ...) (lambda (c) (* c c)))`

31. Section 1.3.2 in the text discusses how to use lambdas in place of locally defined variables. Consider doing a straightforward rewrite of this function, replacing its local definitions in the prescribed manner:

```
(define (f x)
    (define a (+ x 1))
    (define b (- x 1))
    (+ (* a a) (* a b) (* b b)) ; action
    )
```

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

(A) 2
(B) 3
(C) 1
(D) 4

32. Continuing with the previous question, what are valid parameter lists for the lambda(s) that wrap the action in a minimal rewrite?

(A) `(lambda (+ a 1) ...)` and `(lambda (- b 1) ...)`
(B) `(lambda (a b) ...)` and `(lambda (+ x 1) (- b 1) ...)`
(C) `(lambda (a) ...)` and `(lambda (b) ...)`
(D) `(lambda (x) ...)`
(E) `(lambda (a b) ...)`
(F) `(lambda (+ a 1) (- b 1) ...)`

33. Continuing with the previous question, what are valid arguments for the lambda(s) that wrap the action in a minimal rewrite?

(A) `((lambda ...) (+ x 1) (- x 1))` and `((lambda ...) a b)`
(B) `((lambda ...) (+ x 1) (- x 1))`
(C) `((lambda ...) (+ a 1))` and `((lambda ...) (+ b 1))`
(D) `((lambda ...) a b)`
(E) `((lambda ...) x)`
(F) `((lambda ...) a)` and `((lambda (..) ..) b)`

34. Choose the correct time and space complexity of this function:

```
(define (g n)
    (define (iter c)
        (cond
            ((< c 2) c)
            (else (+ (iter (- c 1)) (iter (- c 2))))
            )
        )
    (iter n)
    )
```

(A) linear and quadratic
(B) linear and exponential
(C) exponential and linear
(D) quadratic and linear
(E) exponential and exponential
(F) linear and linear

35. Choose the correct time and space complexity of this function:

```
(define (f n)
    (define (iter x)
        (cond
            ((= x n) n)
            (else (* x (iter (+ x 1))))
            )
        )
    (iter 1)
    )
```

(A) quadratic and constant
(B) quadratic and linear
(C) linear and linear
(D) linear and constant
(E) linear and quadratic
(F) constant and linear

36. **T** or **F**: The following function exhibits tree recursion:

```
(define (g a)
    (if (< a 2)
        a
        (+ (g (- a 2)) (- a 1))
        )
    )
```

37. This function:

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

executes in:

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

38. This function:

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

executes in:

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

39. How is Lamé's Theorem used to generate an upper time bound on `gcd(m,n)`? Assume that *gcd* is defined as:

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

that the initial value of m is larger than n, and that `gcd(m,n)` takes k steps.

(A) $n \geq Fib(k) \rightarrow n \geq \frac{\Phi^k}{\sqrt{5}} \rightarrow log(n) \geq k\ log(\Phi) - log(\sqrt{5})$
(B) $n \leq Fib(k) \rightarrow n \leq \frac{\Phi^k}{\sqrt{5}} \rightarrow log(n) \geq k\ log(\Phi) - log(\sqrt{5})$
(C) $n \leq Fib(k) \rightarrow n \leq \frac{k^\Phi}{\sqrt{5}} \rightarrow log(n) \leq \Phi\ log(k) - log(\sqrt{5})$
(D) $n \geq Fib(k) \rightarrow n \geq \frac{k^\Phi}{\sqrt{5}} \rightarrow log(n) \geq \Phi\ log(k) - log(\sqrt{5})$

40. Suppose *gcd* is defined as:

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

The number of remainder operations performed by `(gcd 412 63)` using applicative order evaluation is:

(A) 4
(B) 5
(C) 3
(D) 6
(E) 2
(F) 7

41. Suppose *gcd* is defined as:

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

Assuming a priori knowledge of when to stop (i.e. there is no checking the value of n to see if the recursion should continue), what is the number of remainder operations performed by `(gcd 123 31)` using normal order evaluation? Assume both m and n are evaluated at the end. Note, it is known in advance that there will be three recursive calls.

(A) 3
(B) 9
(C) 8
(D) 6
(E) 7
(F) 2
(G) 5
(H) 4

42. What is Fermat's little theorem in Scheme notation? Assume n is prime and a positive.

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

43. What is a Charmichael number?

(A) numbers that fool Miller-Rabin-based primality predicates
(B) numbers that fool straight Fermat-based primality predicates
(C) prime numbers undetected by straight Fermat-based primality predicates
(D) prime numbers undetected by Miller-Rabin-based primality predicates

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

```
(define (expmod base exp m)
    (% (fast-expt base exp) m))
```

(A) *fast-expt* may overflow in systems with fixed-sized integers
(B) the mod operation should happen before the call to *fast-expt*
(C) *fast-expt* is slow for systems with fixed-sized integers
(D) *fast-expt* already performs the mod operation

45. Consider the following implementation:

```
(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))))
```

What is the recurrence that describes the even case?

(A) T(n) = T(n/2) + O(log n)
(B) T(n) = 2T(n/2) + O(1)
(C) T(n) = 2T(n/2) + O(log n)
(D) T(n) = T(n/2) + O(1)

46. 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))`

47. To what does the following expression evaluate?

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

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

48. To what does the following expression evaluate?

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

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

49. To what does the following expression evaluate?

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

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

50. Consider the lambda calculus:

```
expr : SYMBOL
     | (lambda (SYMBOL) expr)
     | (expr expr)
```

Respectively, the first, second, and third alternatives correspond to:

(A) identifiers, function definitions, function calls
(B) identifiers, function definitions, argument lists
(C) identifiers, function calls, argument lists
(D) identifiers, function calls, function definitions