Exam 1

Name : _____ Username: _____

INSTRUCTIONS

- This exam has 17 questions, for a total of 100 points, and 20 bonus points.
- Questions do not necessarily appear in order of difficulty.
- For the bonus questions, your answer must be completely correct to receive credit.
- Make every effort to complete all the questions on the exam.

Page:	1	2	3	4	5	6	Total
Points:	21	23	18	23	5	10	100
Score:							

1. Consider the expression below

```
((lambda (s) (lambda (s) (lambda (t) (lambda (s) v))))
 (((lambda (t) (lambda (t) s)) (lambda (v) u))
  (lambda (u) (lambda (s) (lambda (s) u)))))
```

(a) (5 points) List the variables that *occur free* in the expression above.

(a) _____

(b) (5 points) List the variables that *occur bound* in the expression above.

(b) _____

2. (5 points) Consider the following expressions:

```
((lambda (b)
   (lambda (a)
     ((lambda (b) (b a))
      (lambda (a) (a b)))))
 (lambda (e)
   (lambda (d)
     ((lambda (e) (e d))
      (lambda (f) (e e))))))

((lambda (_____)
   (lambda (_____)
     ((lambda (_____) (_____ _____))
      (lambda (_____) (_____ _____)))))
 (lambda (_____)
   (lambda (_____)
     ((lambda (_____) (_____ _____))
      (lambda (_____) (_____ _____))))))
```

If we were to create an expression equivalent to the first expression by filling in the blanks above using the smallest number of distinct variables possible, what is the minimal number of variables required?

A. 2

B. 3

C. 4

D. 8

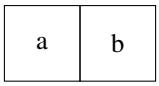
3. (6 points) Consider the expression below, which uses lexical addresses instead of variables.

```
(lambda
  (lambda
    (((lambda 2) (lambda 0))
     (lambda (lambda 3)))))
```

Fill in the blanks in the expression below so that the lambda-calculus expression below would correspond to the expression above. There are multiple correct answers.

```
(lambda (_____)
  (lambda (_____)
    (((lambda (_____) a) (lambda (_____) b))
     (lambda (_____) (lambda (_____) a)))))
```

4. (5 points) Using the model of boxes and arrows to represent cons cells, what does the structure `((a . b) d . ())` look like?

   For reference, the structure `(a . b)` is drawn as: 

5. For each of the following, how many invocations of `cons` does it take to create the Racket value?

   (a) (2 points) `(a (a b) ())`

   (a) _____

   (b) (2 points) `126738443`

   (b) _____

6. (5 points) What is the value of the following Racket expression?

```
(((lambda (x y) (x y))
  (lambda (y) (lambda (y x) (x (x y))))
  (lambda (x) (lambda (x y) (x (y x)))))
 (lambda (x) 2)
 (lambda (x) 3))
```

   6. _____

7. Consider the following definition:

```
(define info
  (lambda (x)
    (match x
      (`(lambda (,x) ,body) `(The x is ,x and the body is ,body)))))
```

   For each of the following, what is the value of the call? (Write ⊥ if the evaluation signals an exception).

   (a) (3 points) `(info '(lambda (z) z))`

   (a) _____

   (b) (3 points) `(info ((lambda (x) x) '(lambda (y) body)))`

   (b) _____

   (c) (3 points) `(info '(f x))`

   (c) _____

8. (8 points) Fill in the expression below to complete the definition of `filter`, a function that takes a predicate `p` and a list `ls`, and returns a list containing only the elements of `ls` that satisfy `p`. Your definition should be a single, naturally-recursive function.

   Consider the following calls below as examples of its usage.

```
> (filter even? '(0 1 2 3 4))
(0 2 4)
> (filter pair? '(0 (1) 2 (3) 4))
((1) (3))

(define filter
  (lambda (p ls)









                                                    ))
```

9. Consider the definitions below of `maxls` and `mystery`.

```
(define maxls
  (lambda (ls)
    (cond
      ((null? ls) 0)
      (else (max (car ls) (maxls (cdr ls)))))))

(define mystery
  (lambda (s)
    (cond
      ((pair? s) (add1 (maxls (map mystery s))))
      (else 0))))
```

   Notice the use of the Racket primitive `max`, that returns the maximum of two numbers, as suggested by its usage in the call below:

```
> (maxls '(1 8 2 7))
8
```

   For the following two parts, what is the value of the call? (Hint: it is generally easier to gain a conceptual understanding of what a function is doing than to trace something this complicated).

   (a) (5 points) `(mystery '(a (b)))`

   (a) _____

   (b) (5 points) `(mystery '((a b) (((c (()))))) e))`

   (b) _____

10. (5 points) Consider the function `foo` below:

```
(define foo
  (lambda (a b)
    (cond
      ((= a b) a)
      ((> a b) (foo (- a b) b))
      (else (foo a (- b a))))))
```

Which of the following is a better name for `foo`?

A. least common multiple

B. set difference

C. greatest common divisor

D. binary XOR

11. (5 points) Consider the function `bar` below:

```
(define bar
  (lambda (x ls)
    (cond
      ((null? ls) #f)
      ((bar x (cdr ls)) (bar x (cdr ls)))
      ((eqv? x (car (car ls))) (car ls))
      (else #f))))
```

What is a better name for the the function `bar`?

A. `memv`

B. `occurs?`$*$

C. `reverse-assv`

D. constant `#f` (that is, a function which always returns false)

12. (9 points) Jason took rather poor notes in class and as a result produced the interpreter below that is riddled with errors. It compiles, but is pretty clearly going to raise an exception in the course of evaluating expressions, such as that below.

```
> (foul-of '(((lambda (z) (lambda (y) (add1 z)))  5) 6)
           (lambda (y) (error 'empty-env "unbound-identifier ~s~n" y)))
<sad exception>
```

Mark (cirle, arrow, strikout, annotate, etc) all the errors in the interpreter which will prevent it from correctly evaluating the above expression. You should make no other changes.

```
(define foul-of
  (lambda (exp env)
    (match
      [`(lambda (x) ,body)
       (lambda (a) (foul-of body (lambda (x) (if (eqv? x y) a (envy y)))))]
      [(? (number? exp)) exp]
      [(symbol? ,exp) env exp]
      [`(,rator ,rand) [(foul-of rator env) (foul-of rand)]]
      [`(add1 ,nexp) (add1 (foul-of nexp env))])))
```

13. (4 points) Fill in the blanks in the following definition of `free-count`, which returns the number of free variable occurances in an expression. For example `(free-count '(lambda (x) (lambda (y) (x z))) '())` $\Rightarrow$ `1`.

```
(define free-count
  (lambda (exp env)
    (match exp
      [(? symbol?) (if _________________ 0 1)]
      [`(lambda (,x) ,e) (free-count _________________ _________________)]
      [`(,e1 ,e2) (__________ (free-count e1 env) (free-count e2 env))])))
```

14. (5 points) Consider the following definition of `value-of`:

```
(define value-of
  (lambda (e env)
    (match e
      [(? symbol?)
       (let ((pr (assv e env)))
         (if pr (cdr pr) (error 'env "unbound var~s~n" e)))]
      [`(lambda (,x) ,body) (lambda (a) (value-of body `((,x . ,a) . ,env)))]
      [`(,rator ,rand) ((value-of rator env) (value-of rand env))])))
```

The `let*2` form works like `let*`, except that it binds exactly two variables. The expression representing the second binding may reference the first variable. For example:

```
(let*2 ([x 5]
        [y (+ x 3)])                    ⇒  10
  (+ y 2))
```

The expression `(let*2 ([x1 e1] [x2 e2]) <body>)` expands into the following expression:

```
(let ([x1 e1])
  (let ([x2 e2])
    <body>))
```

Which of the following `match` lines can be added to the above definition of `value-of` to correctly implement `let*2`?

A.
```
[`(let*2 ([,x1 ,e1] [,x2 ,e2]) ,body)
  (let ([e1 (value-of e1 env)] [e2 (value-of e2 env)])
    (value-of body `((,x1 . ,e1) (,x2 . ,e2) . ,env)))]
```

B.
```
[`(let*2 ([,x1 ,e1] [,x2 ,e2]) ,body)
  (let* ([e1 (value-of e1 env)] [e2 (value-of e2 env)])
    (value-of body `((,x1 . ,e1) (,x2 . ,e2) . ,env)))]
```

C.
```
[`(let*2 ([,x1 ,e1] [,x2 ,e2]) ,body)
  (value-of body `((,x1 . ,e1) (,x2 . ,e2) . ,env))]
```

D.
```
[`(let*2 ([,x1 ,e1] [,x2 ,e2]) ,body)
  (let ([env `((,x1 . ,(value-of e2 env)) . ,env)])
    (value-of body `((,x2 . ,e1) . ,env)))]
```

E.
```
[`(let*2 ([,x1 ,e1] [,x2 ,e2]) ,body)
  (let ([e1 (value-of e1 env)])
    (let ([env `((,x1 . ,e1) . ,env)])
      (let ([e2 (value-of e2 env)])
        (value-of body `((,x2 . ,e2) . ,env)))))]
```

15. Consider the following expression:

```
(let ((x 5))
  (let ((y 7))
    (let ((f (lambda (x) (+ x y))))
      (let ((x 2))
        (f x)))))
```

What is the value of the expression ...

(a) (5 points) ... under lexical scope?

(a) _____

(b) (5 points) ... under dynamic scope?

(b) _____

BONUS

16. (10 bonus points) A *stream* is a potentially infinite list. For example `(nums n)` returns a stream containing all integers greater than or equal to `n`.

```
(define nums
  (lambda (n)
    (cons n (lambda () (nums (add1 n))))))
```

In the space below, define a function `front` that takes a number `n` and a stream `s` and returns a list of the first `n` elements of `s`. For example, `(front 5 (nums 1))` returns `(1 2 3 4 5)`.

17. (10 bonus points) Here are the definitions of our standard `length` function and a new function called `Z`.

```
(define length
  (lambda (ls)
    (cond
      [(null? ls) 0]
      [else (add1 (length (cdr ls)))])))

(define Z
  (lambda (f)
    ((lambda (x) (f (lambda (y) ((x x) y))))
     (lambda (x) (f (lambda (y) ((x x) y)))))))
```

In the space below, define a function `my-length` such that `(Z my-length)` operates similarly to `length`, but uses no explicit recursion. For instance, `((Z my-length) '(a b c d))` should evaluate to `4`.