Exam 1

Name : ____________________ Username: ____________________

INSTRUCTIONS

- This exam has 15 questions, for a total of 100 points and 20 bonus points.
- Questions do not necessarily appear in order of difficulty.
- Make every effort to complete all the questions on the exam. For the bonus questions, your answer must be completely correct to receive credit.

Page:	1	2	3	4	5	6	Total
Points:	14	12	17	29	14	14	100
Score:							

1. (9 points) An unsuspecting Racket programmer has written the following expressions, some of which unfortunately result in errors when evaluated. In which expression(s) must at least one occurrence of `let` be replaced by `letrec` in order to avoid an exception? Mark **all** definitions that must be changed.

```
○ (let ([f (lambda (n)
             (if (zero? n)
                 1
                 (* n (f (sub1 n)))))])
    (f 5))

○ (let ([f (lambda (f)
             (lambda (n)
               (if (zero? n)
                   1
                   (* n ((f f) (sub1 n))))))])
    ((f f) 5))

○ (let ([is-even? (lambda (n)
                    (if (zero? n)
                        #t
                        (is-odd? (sub1 n))))]
        [is-odd? (lambda (n)
                   (if (zero? n)
                       #f
                       (is-even? (sub1 n))))])
    (is-odd? 5))
```

2. (5 points) When implementing **sub1**, we added a line to our interpreter. The line is reproduced below.

```
[`(sub1 ,nexp) (sub1 (val-of nexp env))]
```

We could have, instead of adding a **sub1** line to the interpreter, added a binding of the symbol **sub1** to Racket's **sub1** function in the initial environment, and evaluated expressions in that initial environment, as demonstrated below.

```
> (define init-env
    (lambda ()
      (lambda (y)
        (if (eqv? y 'sub1)
          sub1
          (error 'init-env "unbound␣identifier␣~s~n" y)))))
> (val-of '((lambda (x) (sub1 x)) 5) (init-env))
4
```

What might be a good reason to make the latter choice over the former? (Choose the *best* answer)

A. Racket's **sub1** is faster.
B. **sub1** could then be passed as an argument to a function.
C. **sub1** could then be used as a function of one or more arguments.
D. We no longer have to evaluate **sub1**'s argument before the function is applied.
E. It eliminates a free variable.

3. Read the descriptions of the **take-while** and **drop** functions. Based on those descriptions, finish defining the naturally recursive implementations of the functions and then the CPSed versions of them.

- **take-while** is a function that takes two arguments: a predicate function and a list. Remember, a predicate function takes one input argument (an element of the input list in this case) and returns a boolean. take-while goes from the beginning of the input list and returns its elements in a new list while the predicate holds true.

  Examples:
  - `(take-while (lambda (x) (>= x 5)) '(3 4 5 4 3)) => '(3 4)`
  - `(take-while odd? '(1 3 5 4 3)) => '(1 3 5)`

  - (take-while odd? '(1 3 5 1 3)) => '(1 3 5 1 3)
  - (take-while (**lambda** (x) (**or** (eqv? x 'a) (eqv? x 'b)) '(a b b a d a b c)) => '(a b b a)

- The function **drop** takes two arguments: a natural number and a list. It returns the elements of the input list after skipping over a certain number of elements from the beginning. The number of elements that were skipped over equals the input natural number.

  Examples:

  - (drop 0 '(3 4 5 4 3)) => '(3 4 5 4 3)
  - (drop 10 '(1 3 5 4 3)) => '()
  - (drop 5 '(a b b a d a b c)) => '(a b c)

(a) (6 points) Naturally recursive take-while:

```
(define take-while
  (lambda (p? ls)



                                                  ))
```

(b) (6 points) CPSed take-while:

```
(define take-while-cps
  (lambda (p? ls k)



                                                  ))
```

(c) (6 points) Naturally recursive drop:

```
(define drop
  (lambda (n ls)









                                                  ))
```

(d) (6 points) CPSed drop:

```
(define drop-cps
  (lambda (n ls k)









                                                  ))
```

For the following two questions, rewrite each expression, replacing each variable reference with an integer representing that variable's lexical address. Use -1 for free variables.

4. (5 points)

```
(lambda (e)
  (lambda (g)
    ((lambda (e)
       ((g (lambda (h) g)) (lambda (e) (lambda (h) h))))
     (lambda (h) ((lambda (e) (lambda (e) e)) f)))))
```

```
(lambda
  (lambda
    ((lambda
       ((______ (lambda ______)) (lambda (lambda ______))))
     (lambda ((lambda (lambda ______)) ______)))))
```

5. (7 points)

```
(lambda (b)
  (lambda (a)
    ((lambda (n) (a n))
     (lambda (a)
       (b (lambda (i) ((k i) (n i))))))))
```

```
(lambda
  (lambda
    ((lambda (______ ______))
     (lambda
       (______ (lambda ((______ ______) (______ ______))))))))
```

6. Consider the following expression:

```
(let ([g (lambda (y) (f (+ 6 y)))])
  (let ([f (lambda (z) (+ 7 z))])
    (g 5)))
```

What is the value of the expression ...

(a) (3 points) ... under lexical scope?

(a) ______________

(b) (3 points) ... under dynamic scope?

(b) ______________

7. 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.

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

8. For each of the following, how many invocations of **cons** does it take to create the racket value?

(a) (3 points) `'(() a (b c) ())`

(a) ______________

(b) (3 points) `'((5 4) ((4 3) (3)))`

(b) ______________

9. (6 points) We saw in class that **match** provides an easy way to match on data structures and bind their parts, such as in the expression below.

```
(match expr
  (`(tag . ,d) ...₁)
  (else ...₂)
```

Suppose we didn't have **match**. We could use Racket predicates and let bindings to do the work that **match** is doing.

Fill in the blanks below to complete an expression equivalent to the one above that does not use match.

```
(cond
  (


   (let (


                                                   )
      ...₁))
  (else ...₂))
```

10. Recall that for all the four call-by interpreters, the number line of the interpreter is implemented in the same way and is shown below.

```
 (match exp
   ...
   [`,n #:when (number? n) n]
   ...)
```

During the evaluation below

```
(valof '((lambda (x) (* x x)) 5) (empty-env))
```

how many times is the right-hand side of the number line evaluated when -

(a) (2 points) valof is a call-by-value interpreter?

(a) ______________

(b) (2 points) valof is a call-by-reference interpreter?

(b) ______________

(c) (2 points) valof is a call-by-name interpreter?

(c) ______________

(d) (2 points) valof is a call-by-need interpreter?

(d) ______________

11. (6 points) Consider this partially representation-independent interpreter:

```
(define value-of
  (lambda (e env)
    (match e
      [`,x #:when (symbol? x) (apply-env env x)]

      [`(let ([,x ,e]) ,body)
         (let ([a (value-of e env)])
           (value-of body (lambda (y) (if (eq? y x) a (env y)))))]

      [`(lambda (,x) ,body)
         (lambda (a) (value-of body (extend-env x a env)))]

      [`(,rator ,rand) ((value-of rator env) (value-of rand env))])))
```

Also consider this application of the interpreter:

```
(value-of '((lambda (x) x) (lambda (y) y))
  (lambda (y) (error 'apply-env "Unbound␣variable␣~s" y)))
```

Circle each place where the interpreter is **not** representation-independent with respect to environments **and** closures. Mark the environment representations you circle with an *e*, and the closure representations with a *c*.

12. (4 points) Describe what a closure is in one sentence.

13. `let/cc` evaluation.

(a) (3 points) Evaluate the following

```
(let/cc k (k (+ (k (+ 2 (k 3)))
                4)))
```

A. 5
B. 4
C. 3
D. infinite loop

(b) (1 point) Evaluate the following assuming plus evaluates its arguments from left to right

```
(let ([k (let/cc k (k (+ (+ (k (car (k (cdr `(,k 10)))))
                            (k (lambda (k) k)))
                         (k k))))])
  (if (number? k) k (k k)))
```

A. 10
B. 20
C. error
D. infinite loop

BONUS

14. (10 bonus points) Here is a call-by-name interpreter:

```
(define value-of
  (lambda (e env)
    (match e
      [`,y #:when (symbol? y) ((env y))]
      [`(lambda (,x) ,body) (lambda (a)
                              (value-of body
                                        (lambda (y)
                                          (if (eqv? x y)
                                              a
                                              (env y)))))]
      [`(,rator ,x) #:when (symbol? x) ((value-of rator env) (env x))]
      [`(,rator ,rand) ((value-of rator env)
                        (lambda ()
                          (value-of rand env)))])
```

Revise this interpreter to implement a call-by-need interpreter by making changes to the right-hand side of **only one** *match* clause. You only need to write in the one line of the interpreter you're modifying. There are multiple correct solutions. Again, you may modify the right-hand side of **only one** *match* clause.

```
(define value-of
  (lambda (e env)
    (match e
      [`,y #:when (symbol? y)


                                                          ]
      [`(lambda (,x) ,body)


                                                          ]
      [`(,rator ,x)


                                                          ]
      [`(,rator ,rand)




                                                          ])))
```

15. (10 bonus points) Here is the definition of a function `val-of` which is a dynamic scope interpreter. Observe that this interpreter is not representation independent and uses higher order function representation for both its environment and closure. However, the **`letrec`** pattern case uses a function `ext-rec-env` to extend the recursive environment.

```
(define val-of
  (lambda (e env)
    (match e
      [`,b #:when (boolean? b) `(,b ,env)]
      [`,n #:when (number? n) `(,n ,env)]
      [`(cons ,a ,d)
       (match-let* ([`(,res-of-a ,env) (val-of a env)]
                    [`(,res-of-d ,env) (val-of d env)])
         `(,(cons res-of-a res-of-d) ,env))]
      [`(car ,l)
       (match-let ([`(,res-of-l ,env) (val-of l env)])
         `(,(car res-of-l) ,env))]
      [`(cdr ,l)
       (match-let ([`(,res-of-l ,env) (val-of l env)])
         `(,(cdr res-of-l) ,env))]
      [`(quote ,e) `(,e ,env)]
      [`(null? ,l)
       (match-let ([`(,res-of-l ,env) (val-of l env)])
         `(,(null? res-of-l) ,env))]
      [`(+ ,nexp1 ,nexp2)
       (match-let* ([`(,res-of-nexp1 ,env) (val-of nexp1 env)]
                    [`(,res-of-nexp2 ,env) (val-of nexp2 env)])
         `(,(+ res-of-nexp1 res-of-nexp2) ,env))]
      [`(if ,t ,c ,a)
       (match-let ([`(,res-of-t ,env) (val-of t env)])
         (if res-of-t (val-of c env) (val-of a env)))]
      [`(zero? ,nexp)
       (match-let ([`(,res-of-nexp ,env) (val-of nexp env)])
         `(,(zero? res-of-nexp) ,env))]
      [`(sub1 ,nexp)
       (match-let ([`(,res-of-nexp ,env) (val-of nexp env)])
         `(,(sub1 res-of-nexp) ,env))]
      [`(letrec ,1/2-closures ,b)
       (val-of b (ext-rec-env 1/2-closures env))]
      [`,y #:when (symbol? y) `(,(env y) ,env)]
      [`(lambda (,x) ,body)
       #:when (symbol? x)
       `(,(lambda (arg env)
            (val-of body (lambda (y)
                           (cond
                             [(eqv? y x) arg]
                             [else (env y)]))))
         ,env)]
      [`(,rator ,rand)
       (match-let* ([`(,res-of-rator ,env) (val-of rator env)]
                    [`(,res-of-rand ,env) (val-of rand env)])
         (res-of-rator res-of-rand env))])))
```

Here is an example invocation of val-of:

```
(val-of '(letrec ([map (lambda (f)
                          (lambda (ls)
                            (if
                             (null? ls) '()
                             (cons
                              (f (car ls))
                              ((map f) (cdr ls))))))]
                   [add1 (lambda (n)
                           (+ n 1))])
           ((lambda (ls)
              ((map (lambda (x) (cons (add1 x) ls)))
               ls))
            (cons 1 (cons 2 (cons 3 '())))))
        (lambda (y)
          (error "Not␣a␣program!␣Free␣variable:␣" y)))
```

Define `ext-rec-env` so that `val-of` still behaves like a normal dynamic scope interpreter.

```
(define ext-rec-env
  (lambda (1/2-closures env)



                                                        ))
```

Appendix: Call by Value, Call by Reference Interpreters

Note: You may assume the usual lines for primitives (numbers, +, etc.) are included where `...` appears in these definitions.

Call by Value

```
(define value-of-cbv
  (lambda (exp env)
    (match exp
      [`,y #:when (symbol? y) (unbox (env y))]
        ...
      [`(begin2 ,e1 ,e2) (begin (value-of-cbv e1 env) (value-of-cbv e2 env))]
      [`(set! ,x ,e) (set-box! (env x) (value-of-cbv e env))]
      [`(lambda (,x) ,body)
          (lambda (a)
            (value-of-cbv body (lambda (y) (if (eqv? x y) a (env y)))))]
      [`(,rator ,rand)
        ((value-of-cbv rator env) (box (value-of-cbv rand env)))])))
```

Call by Reference

```
(define value-of-cbr
  (lambda (exp env)
    (match exp
      [`,y #:when (symbol? y) (unbox (env y))]
        ...
      [`(begin2 ,e1 ,e2) (begin (value-of-cbr e1 env) (value-of-cbr e2 env))]
      [`(set! ,x ,e) (set-box! (env x) (value-of-cbr e env))]
      [`(lambda (,x) ,body)
          (lambda (a)
            (value-of-cbr body (lambda (y) (if (eqv? x y) a (env y)))))]
      [`(,rator ,x) #:when (symbol? x) ((value-of-cbr rator env) (env x))]
      [`(,rator ,rand)
        ((value-of-cbr rator env) (box (value-of-cbr rand env)))])))
```

Appendix: Call by Name, Call by Need Interpreters

Note: You may assume the usual lines for primitives (numbers, +, etc.) are included where `...` appears in these definitions.

Call by Name

```
(define value-of-cbname
  (lambda (exp env)
    (match exp
      [`,y #:when (symbol? y) ((unbox (env y)))]
       ...
      [`(lambda (,x) ,body)
          (lambda (a)
            (value-of-cbname body (lambda (y) (if (eqv? x y) a (env y)))))]
      [`(,rator ,x) #:when (symbol? x) ((value-of-cbname rator env) (env x))]
      [`(,rator ,rand)
        ((value-of-cbname rator env)
         (box (lambda () (value-of-cbname rand env))))])))
```

Call by Need

```
(define value-of-cbneed
  (lambda (exp env)
    (match exp
      [`,y #:when (symbol? y)
        (let ([b (env y)])
          (let ([v ((unbox b))])
            (begin (set-box! b (lambda () v)) v)))]
      ...
      [`(lambda (,x) ,body)
          (lambda (a)
            (value-of-cbneed body (lambda (y) (if (eqv? x y) a (env y)))))]
      [`(,rator ,x) #:when (symbol? x) ((value-of-cbneed rator env) (env x))]
      [`(,rator ,rand)
        ((value-of-cbneed rator env)
         (box (lambda () (value-of-cbneed rand env))))])))
```