```
#lang racket

(define (literal-compare x y)
    (if (equal? x y) x (list `if `% x y)))

(define (list-compare x y)
    (cond
        [(equal? x `()) `()] ; base case

        [(equal? (car x) (car y)) (cons (car x) (list-compare (cdr x) (cdr y)))] ;
both match

        [else (cons (expr-compare (car x) (car y)) (list-compare (cdr x) (cdr y)))]
; mismatch
    )
)

(define (lambda? x) (and (member x '(lambda λ)) #t))

(define (substitute-vars expr old-vars new-vars) ; helper to substitute the args
    (cond
        [(symbol? expr) (let ((pos (index-of old-vars expr))) (if pos (list-ref
new-vars pos) expr))]

        [(pair? expr) (cons (substitute-vars (car expr) old-vars new-vars)
(substitute-vars (cdr expr) old-vars new-vars))]

        [else expr]
    )
)

(define (lambda-compare x y) ; (car) is the arg list (cadr) is the lambda function
body
    (let* ((new-formals (map (lambda (x-var y-var) (if (equal? x-var y-var) x-var
        (string->symbol (string-append (symbol->string x-var) "!" (symbol->string
y-var))))) (cadr x) (cadr y)))

        (x-body-subst (substitute-vars (caddr x) (cadr x) new-formals))
        (y-body-subst (substitute-vars (caddr y) (cadr y) new-formals))
        (new-body (expr-compare x-body-subst y-body-subst)))

        (cons (if (or (equal? (car x) 'λ) (equal? (car y) 'λ)) 'λ 'lambda) (list
new-formals new-body))
        ; ^ lambda symbol check
    )
)

(define (expr-compare x y)
    (cond
        [(equal? x y) x] ; identical
```

```
        [(and (boolean? x) (boolean? y)) (if x `% `(not %))] ; 2 booleans

        [(or (not (list? x)) (not (list? y))) (list `if `% x y)] ; list and literal

        [(not (equal? (length x) (length y))) (list `if `% x y)]
        ; ^ different length lists or two different elems

        [(or (equal? (car x) `quote) (equal? (car y) `quote)) (list `if `% x y)];
quote statement

        [(not (equal? (equal? (car x) `if) (equal? (car y) `if)))  (list `if `% x
y)]
        ; ^  one sided if statement

        [(xor (lambda? (car x)) (lambda? (car y))) (list `if `% x y)]
        ; ^ one sided lambda

        [(and (lambda? (car x)) (lambda? (car y))) (lambda-compare x y)]

        ; lambda statement

        [else (list-compare x y)] ; everything else
    )
)

(define (test-expr-compare x y)
    (define ns (make-base-namespace)) ; namespace to evaluate expressions

    (define x-act (eval x ns))
    (define y-act (eval y ns))

    (define x-chk (expr-compare x y))

    (define x-chk-t (eval `(let ((% #t)) ,x-chk) ns))

    (define x-chk-f (eval `(let ((% #f)) ,x-chk) ns))

    (and (equal? x-act x-chk-t) (equal? y-act x-chk-f))
)

(define test-expr-x '((lambda (x y) (if x (list y (lambda (a b) (f a b))) (cons y
((lambda (a c) (g a c)) 1 2)))) #t 42))

(define test-expr-y '((λ (x z) (if z (list z (lambda (a d) (f a d))) (list x
((lambda (a e) (h a e)) 1 3)))) #f 99))
```