```
;;main logic of program
(define expr-compare
    (lambda (x y)
        (if (or (null? x) (null? y))
            ;;base case
            '()
            (cond
                ((and (list? x) (list? y))
                    (let ((headx (car x)) (heady (car y)) (tailx (cdr x)) (taily (cdr y)))
                        (cond
                            ((not (eqv? (length x) (length y))) (sym-handler x y))
                            ((equal? x y) x)
                            ;;are either a let expression
                            ((or (equal? headx 'let) (equal? heady 'let))
                                (if (and (equal? headx 'let) (equal? heady 'let))
                                    (let ((dec (let-handler tailx taily (car tailx) (car taily))))
                                        (cons 'let (expr-compare (car dec) (cdr dec))))
                                    ;;if both are not let expressions then they cannot be considered similar
                                    (sym-handler x y))
                            )
                            ;;are either lambda expressions
                            ((or (equal? headx 'lambda) (equal? heady 'lambda))
                                (if (and (equal? headx 'lambda) (equal? heady 'lambda))
                                    (let ((dec (lambda-handler tailx taily (car tailx) (car taily))))
                                        (cons 'lambda (expr-compare (car dec) (cdr dec))))
                                    ;;if both are not lambda then can't be similar
                                    (sym-handler x y))
                            )
                            ;;are either quote expressions
                            ((or (equal? headx 'quote) (equal? heady 'quote))
                                (cond
                                ((and (equal? headx 'quote) (equal? heady 'quote))
                                    `(if % ',(car tailx) ',(car taily)))
                                ((and (equal? headx 'quote) (not (equal? heady 'quote)))
                                    `(if % ',(car tailx) ,y))
                                ((and (not (equal? headx 'quote)) (equal? heady 'quote))
                                    `(if % ,x ',(car taily)))
                                ))
                            ((or (and (equal? headx 'if) (not (equal? heady 'if))) (and (equal? heady 'if) (not (equal?
headx 'if))))
                                (sym-handler x y))
                            (else (cons (expr-compare headx heady) (expr-compare tailx taily))))))
                (else (sym-handler x y))
            )
        )
```

```
    )
)

;;used when args are determined to be symbol, primary base case
(define sym-handler
    (lambda(x y)
        (cond
            ((equal? x y) x)
            ((and (boolean? x) (boolean? y))
                (cond
                    ((and x (not y)) '%)
                    ((and (not x) y) '(not %))))
            (else `(if % ,x ,y))
        )
    )
)



;;returns a symbol x!y for given arguments x and y
(define bang-me
    (lambda (x y)
        (string->symbol (string-append (symbol->string x) "!" (symbol->string y)))))

;;scans through lambda formal and replaces variables when needed in argument list and formal
(define lambda-handler
    (lambda (x y arglist1 arglist2)
        (if
            (or (null? arglist1) (null? arglist2) (not (eqv? (length arglist1) (length arglist2)))) (cons x y) ;change
this?
            (let ((arg1 (car arglist1)) (arg2 (car arglist2)))
                (if (eqv? arg1 arg2)
                    (lambda-handler x y (cdr arglist1) (cdr arglist2))
                    (let ((rep (bang-me arg1 arg2)))
                        (lambda-handler (replace x arg1 rep) (replace y arg2 rep) (cdr arglist1) (cdr arglist2))
                    )
                )
            )
        )
    )
)

;;handles two let expressions
(define let-handler
    (lambda (x y bindings1 bindings2)
        (let-body-handler x y (let-args bindings1) (let-args bindings2) bindings1 bindings2)
```

```
    )
)

;;scans through list of arguments to see if associated positions don't match
;;if arguments don't match, then change in both body and binding of let expressions
(define let-body-handler
    (lambda (x y arglist1 arglist2 bindings1 bindings2)
        (if
            (or (null? arglist1) (null? arglist2) (not (eqv? (length arglist1) (length arglist2)))) (cons x y) ;change
this?
            (let ((arg1 (car arglist1)) (arg2 (car arglist2)))
                (if (eqv? arg1 arg2)
                    (let-body-handler x y (cdr arglist1) (cdr arglist2) bindings1 bindings2)
                    (let ((rep (bang-me arg1 arg2)))
                        (let ((repbind1 (let-bindings bindings1 arg1 rep)) (repbind2 (let-bindings bindings2 arg2 rep)))
                            (let-body-handler
                                (cons repbind1 (replace (cdr x) arg1 rep))
                                (cons repbind2 (replace (cdr y) arg2 rep))
                                (cdr arglist1) (cdr arglist2) repbind1 repbind2)
                        )
                    )
                )
            )
        )
    )
)

;;returns let binding in lambda formal form
(define let-args
    (lambda (arglist)
        (if (null? arglist)
            '()
            (cons (car (car arglist)) (let-args (cdr arglist)))
        )
    )
)

;;replaces variables being bound in binding of let expression
(define let-bindings
    (lambda (bindings targ rep)
        (if (null? bindings) '()
            (let ((head (car bindings)) (tail (cdr bindings)))
                (if (eqv? (car head) targ)
                    (cons (cons rep (cdr head)) (let-bindings tail targ rep))
                    (cons head (let-bindings tail targ rep))
                )
```

```
            )
        )
    )
)


;;finds and replaces values
;;special cases for finding lets and lambdas
;; follows shadow/scope rules
(define replace
    (lambda (x targ rep)
        (if (null? x) '()
        (let ((head (car x)) (tail (cdr x)))
            (cond
                ((list? head) (cons (replace head targ rep) (replace tail targ rep)))
                ((eqv? head targ) (cons rep (replace tail targ rep)))
                ((equal? head 'lambda)
                    (if (in-formal (car tail) targ)
                    x
                    (cons head (cons (car tail) (replace (cdr tail) targ rep)))))
                ((equal? head 'let)
                    (if (in-bindings (car tail) targ)
                    (cons head (cons (replace-let (car tail) targ rep) (cdr tail)))
                    (cons head (replace (cons (replace-let (car tail) targ rep) (cdr tail)) targ rep))))
                ((equal? head 'quote) x)
                (else (cons head (replace tail targ rep)))
                )
            )
        )
    )
)

;;replaces values used to define bindings in let expression
(define replace-let
    (lambda (x targ rep)
        (if (null? x) '()
        (if (eqv? (car (cdr (car x))) targ)
            (cons (cons (car (car x)) (cons rep '())) (replace-let (cdr x) targ rep))
            (cons (car x)  (replace-let (cdr x) targ rep))
        ))))

;;is variable defined in formal for lambda expression
;; use to determine of shadowing occurs
(define in-formal
    (lambda (x targ)
        (if (null? x) #f
```

```
            (if (eqv? (car x) targ) #t
                (in-formal (cdr x) targ)))))

;;is variable defined in bindings for let expression?
;;converts to arglist and uses in-formal
(define in-bindings
    (lambda (x targ)
        (in-formal (let-args x) targ)))


;;function defined in spec
;; defines a masterx variable to be the result of x
;; defines a mastery variable to be the result of y
;; defines true and false for comparison based on %
(define test-expr-compare
    (lambda (x y)
        (let ((masterx (eval x)) (mastery (eval y))
                (true (eval `(let ((% #t)) ,(expr-compare x y))))
                (false  (eval `(let ((% #f)) ,(expr-compare x y)))))
            (and (equal? masterx true) (equal? mastery false)))))

;;first part of test case
(define test-expr-x
    '(list (*
            (let ((x 1) (y 7))
                (lambda (x y)
                    ;; testing shadowing and scope
                    (* x y))
                    x y)
                ((lambda (z g h)
                    ;; testing handling of if speecial form vs procedure
                    (if (> z g) (+ g h) (+ z h)))
                    13 7 2))
        (let ((a "b") (b "9"))
            (let ((b a) (a b))
                ;; testing shadowing and scope
                ((lambda (a b) (quote a)) b a)))
        ((lambda (x)
            ;;test not % and % special case
            (if (or #f #t) #f x)) #t)))


;;second part of test case
(define test-expr-y
    '(list
        (*
```

```
    (let ((b 0) (a 7))
        (lambda (y x)
            ;; testing shadowing and scope
            (+ x a))
            b a)
        ((lambda (h i j)
            ;; testing handling of if speecial form vs procedure
            (* h i j))
            1 3 3))
;;test bindings for both let and lambda
(let ((x "b") (y "9"))
    (let ((b x) (a y))
        ;; testing shadowing and scope
        ((lambda (x y) (quote a)) b y)))
((lambda (x)
    ;;test not % and % special case
    (if (or #f #f) #t x)) #f)))
```