```
; Simple xor logical operator implemented for the if special case
(define (xor a b) (not (boolean=? a b)))

; For constant literals, variable references, two arguments that aren't
the same type, and lists of different sizes
(define (simple-compare x y) (if (equal? x y) x (list 'if '% x y)))

; For lists and procedure calls
(define (list-compare x y) (if (or (equal? x '()) (equal? y '())) '()
                            (cons (expr-compare (car x) (car y)) (list-
compare (cdr x) (cdr y)))))

; For boolean special case
(define (boolean-compare x y) (if x (if y #t '%) (if y (list 'not '%)
#f)))

; Check if they use the same lambda keyword or one of each
(define (auxiliary-lambda x y) (cond
                                  ; Formal arguments are lists of equal
length
                                  [(and (list? (cadr x)) (list? (cadr y))
(equal? (length (cadr x)) (length (cadr y))))
                                   (if (equal? (car x) (car y)) (cons (car
x) (lambda-list-compare (cdr x) (cdr y)))
                                        (cons 'λ (lambda-list-compare (cdr
x) (cdr y))))]
                                  ; Formal arguments are constant
literals, or other same types
                                  [(not (or (list? (cadr x)) (list? (cadr
y))))
                                   (if (equal? (car x) (car y)) (cons (car
x) (lambda-simple-compare (cdr x) (cdr y)))
                                        (cons 'λ (lambda-simple-compare
(cdr x) (cdr y))))]
                                  ; Formal arguments are lists of
different length or different types
                                  [else (list 'if '% x y)]))

; Handle creation of binding
(define (make-bind x y) (string->symbol
                          (string-append (symbol->string x)
                                         (string-append "!" (symbol-
>string y)))))

; Handle comparison of arguments
(define (lambda-arg-compare x y) (if (equal? x y) x (make-bind x y)))

; Handle arguments if they're a list
(define (lambda-arglist-compare x y) (if (or (equal? x '()) (equal? y
'())) '()
                            (cons (lambda-arg-compare (car x) (car y))
(lambda-arglist-compare (cdr x) (cdr y)))))

; Creates an association list for arguments to a lambda form with the !
expression
(define (auxiliary-assoc x y bindings x-bind y-bind) (if (or (equal? x
'()) (equal? y '()) (equal? bindings '())) (list x-bind y-bind)
                                                (auxiliary-assoc (cdr x)
(cdr y) (cdr bindings) (cons (list (car x) (car bindings)) x-bind) (cons
(list (car y) (car bindings)) y-bind))))
```

```
; Uses the association list for each expression to replace bindings,
nested lambda forms are exempt from replacement
(define (auxiliary-bind-replace expr bindings) (if (equal? expr '()) '()
                                                   (if (list? (car expr))
                                                       (if (or (equal?
'lambda (caar expr)) (equal? 'λ (caar expr)))
                                                           (cons (car
expr) (auxiliary-bind-replace (cdr expr) bindings))
                                                           (cons
(auxiliary-bind-replace (car expr) bindings) (auxiliary-bind-replace (cdr
expr) bindings)))
                                                       (cond
                                                         [(equal? (assq
(car expr) bindings) #f) (cons (car expr) (auxiliary-bind-replace (cdr
expr) bindings))]
                                                         [else (cons
(cadr (assq (car expr) bindings)) (auxiliary-bind-replace (cdr expr)
bindings))]))))

; After replacing all bindings, pass the expressions to the regular expr-
compare for processing
; Note undefined behavior if the body of lambda is more than a single
expression: this implementation will not diff
; the entirety of the lambda, but process multiple expression bodies as
if they were valid
(define (lambda-body-compare x y x-bindings y-bindings) (expr-compare
(auxiliary-bind-replace x x-bindings) (auxiliary-bind-replace y y-
bindings)))

; Auxiliary function to take pieces of output and pack into one list
(define (auxiliary-packing x y bindings bind-list) (cons bindings
(lambda-body-compare x y (car bind-list) (cadr bind-list))))

; Auxiliary function to obtain package for output
(define (auxiliary-package x-args y-args binding-pass x-body y-body
bindings) (auxiliary-packing x-body y-body bindings (auxiliary-assoc x-
args y-args binding-pass '() '())))

; This is pretty bad code, since I have two redundant calls, but it
works, can fix later
; Lambda form decomposition if arguments are a list
(define (lambda-list-compare x y) (auxiliary-package (car x) (car y)
(lambda-arglist-compare (car x) (car y)) (cdr x) (cdr y) (lambda-arglist-
compare (car x) (car y))))

; Lambda form decomposition if arguments are not a list
(define (lambda-simple-compare x y) (auxiliary-package (list (car x))
(list (car y)) (list (lambda-arg-compare (car x) (car y))) (cdr x) (cdr
y) (lambda-arg-compare (car x) (car y))))

; expr-compare, works like an if tree, checking each special case and
delegating to appropriate procedure calls
(define (expr-compare x y) (if (and (boolean? x) (boolean? y)) (boolean-
compare x y)
                               ; Check that arguments are lists of equal
size
                               (if (and (list? x) (list? y) (equal?
(length x) (length y)))
                                   ; Check the quote datum special case
                                   (if (or (equal? 'quote (car x))
(equal? 'quote (car y))) (simple-compare x y)
                                       ; Check for lambda special case:
both inputs are lambda
                                       (if (and (or (equal? 'lambda (car
x)) (equal? 'λ (car x)))
                                                (or (equal? 'lambda (car
y)) (equal? 'λ (car y)))) (auxiliary-lambda x y)
                                           ; Check for if special case
                                           (if (xor (equal? 'if (car x))
(equal? 'if (car y))) (simple-compare x y)
                                               ; Otherwise, it's just a
procedure call, normal, list, or double if
                                               (list-compare x y))))
                                   ; For constant literals, variable
references, etc arguments
                                   (simple-compare x y))))

; Test implementation
(define (test-expr-compare x y) (and
                                 (equal? (eval x) (eval (list 'let '((%
#t)) (expr-compare x y))))
                                 (equal? (eval y) (eval (list 'let '((%
#f)) (expr-compare x y))))
                                 ))

; X test
(define test-expr-x '(list
                      ; numbers
                      42
                      42
                      ; booleans
                      #t
                      #t
                      #f
                      #f
                      ; different types
                      12
                      #f
                      ; strings
                      "talking"
                      "walking"
                      ; variable references
                      'a
                      'a
                      'a
                      'a
                      ; lists/procedure calls
                      '(f a)
                      '(cons a b)
                      '(cons a b)
                      '(a b (c d) e (f g))
                      ; quote
                      ''(a k)
                      '(quote (a k))
                      '(quote (a k))
                      ; if
                      '(if x y z)
                      '(if x y z)
                      '(if x (if x y z) z)
                      '(if x y z)
                      ; lambda
                      '(lambda a b)
                      '(lambda a a)
                      '(lambda (a b) a b)
                      '(λ (a b) a b)
                      '(lambda (a b) a b)
                      '(λ (a b) a b)
                      '(lambda a b)
                      '(lambda (a b) a)
                      '(lambda (a b) (a b))
                      '(+ #t (lambda (a b) (eq? a ((λ (a b c) ((λ (a b c)
(a b)) b a c)) a (lambda (a) b)) b)))
                      ))

; Y test
(define test-expr-y '(list
                      ; numbers
                      42
                      24
                      ; booleans
                      #t
                      #f
                      #f
                      #t
                      ; different types
                      #f
                      12
                      ; strings
                      "talking"
                      "running"
                      ; variable references
                      'a
                      'b
                      '(a)
                      '(a b)
                      ; lists/procedure calls
                      '(f a b)
                      '(cons a b)
                      '(cons a c)
                      '(a r (d c) s (f t))
                      ; quote
                      ''(k a)
                      '(quote (k a))
                      '(quoth (k a))
                      ; if
                      '(if x y z)
                      '(if z y z)
                      '(if x (if x y y) z)
                      '(f a b c)
                      ; lambda
                      '(lambda a b)
                      '(lambda b b)
                      '(lambda (a b) a b)
                      '(λ (a c) a c)
                      '(λ (b a) b a)
                      '(lambda (a c) a a)
                      '(lambda (a) b)
                      '(lambda (a b c) b)
                      '(lambda (a b) (a b c))
```

```
                    '(+ #t (lambda (b a) (eqv? a ((λ (a b c) ((λ (a b
c) (a b)) a b c)) a (lambda (a) b)) c)))
                    ))
```