PDP Test Report for SET12

Test Name: draw+eval-tests
Definitions:

```
        (define (eval p) (map (λ (r) (send r to-expr)) (send (mk-
Program% p) eval)))
        (define (subst r x e) (send (send (mk-Expr% e) subst (mk-Expr%
r) x) to-expr))
        (define (expr->expr/no-var e) (send (mk-Expr% e) to-expr/no-
var))
        (define (expr=? e1 e2) (send (mk-Expr% e1) expr=? (mk-Expr%
e2)))
        (define ARITH-EXPR-PLUS (make-arith '+ (list 123 123)))
        (define ARITH-EXPR-SUBT (make-arith '- (list 123 123)))
        (define ARITH-EXPR-MULT (make-arith '* (list 10.5 10.5)))
        (define ARITH-EXPR-DIV (make-arith '/ (list 125 25)))
        (define ARITHMETIC-EXPR-PROGRAM
          (list ARITH-EXPR-PLUS ARITH-EXPR-SUBT ARITH-EXPR-MULT ARITH-
EXPR-DIV))
        (define CMP-GREATHERTHAN-EXPR (make-cmp '> (list 37 36)))
        (define CMP-LESSTHAN-EXPR (make-cmp '< (list 36 37)))
        (define CMP-EQUALTO-EXPR (make-cmp '= (list 37 37)))
        (define COMPARISON-PROGRAM
          (list CMP-GREATHERTHAN-EXPR CMP-LESSTHAN-EXPR CMP-EQUALTO-
EXPR))
        (define BOOL-AND-EXPR (make-bool 'and (list #t #t #t #t)))
        (define BOOL-OR-EXPR (make-bool 'or (list #t #t #f #f)))
        (define BOOL-AND-FALSE-EXPR (make-bool 'and (list #t #f)))
        (define BOOL-OR-FALSE-EXPR (make-bool 'or (list #f #f)))
        (define BOOLEAN-EXPR-PROGRAM
          (list BOOL-AND-EXPR BOOL-OR-EXPR BOOL-AND-FALSE-EXPR BOOL-
OR-FALSE-EXPR))
        (define NUM1 23)
        (define NUM2 12)
        (define NUM3 23)
        (define BOOL-EXPR-1 (make-bool 'or (list (< NUM1 NUM2) (> NUM1
NUM2))))
        (define BOOL-EXPR-2 (make-bool 'or (list (< NUM1 NUM3) (> NUM1
NUM3))))
        (define EXPR1 (make-arith '+ (list NUM1 NUM2)))
        (define EXPR2 (make-arith '- (list NUM1 NUM3)))
        (define IF-EXPR-EVALUATES-EXPR1 (make-if-exp BOOL-EXPR-1 EXPR1
EXPR2))
        (define IF-EXPR-EVALUATES-EXPR2 (make-if-exp BOOL-EXPR-2 EXPR1
EXPR2))
        (define IF-EXPR-PROGRAM (list IF-EXPR-EVALUATES-EXPR1 IF-EXPR-
EVALUATES-EXPR2))
        (define SQR-DEF
          (make-def 'Square (list 'num1) (make-arith '* (list 'num1
'num1))))
        (define CALL-SQR-DEF (make-call 'Square (list 2)))
        (define LAMBDA
          (make-lam (list 'radius) (make-arith '* (list 2 3.142
'radius))))
        (define CIRCUMFERENCE-CALL (make-call LAMBDA (list 5)))
        (define CIRCUMFERENCE-OF-CIRCLE-PROG (list CIRCUMFERENCE-
CALL))
        (define CHK-DENOM
          (make-def 'checkDenominator (list 'num1) (make-cmp '= (list
'num1 0))))
        (define CHK-DIVIDE-BY-ZERO
          (make-def
           'avoidDivideByZero
           (list 'num1 'num2)
           (make-if-exp
            (make-call 'checkDenominator (list 'num2))
            "Cannot Divide by Zero"
            (make-arith '/ (list 'num1 'num2)))))
        (define CHK-DIVIDE-BY-ZERO_1 (make-call 'avoidDivideByZero
(list 10 5)))
        (define CHK-DIVIDE-BY-ZERO_2 (make-call 'avoidDivideByZero
(list 5 0)))
        (define PROGRAM-CHK-DIVIDE-BY-ZERO
          (list
           CHK-DENOM
           CHK-DIVIDE-BY-ZERO
           CHK-DIVIDE-BY-ZERO_1
           CHK-DIVIDE-BY-ZERO_2))
        (define FINAL-PROGRAM
          (list ARITH-EXPR-PLUS ARITH-EXPR-MULT CALL-SQR-DEF SQR-DEF
#t 365))
        (define TEST-EXPR-FOR-EXPR-WITH-NO-VAR
          (make-lam (list 'num1 'num2) (make-arith '+ (list 'num1
'num2))))
        (define TEST-EXPR-WITH-NO-VAR-RESULT
          (make-lam/no-var (make-arith '+ '((0 0) (0 1)))))
        (define LAMBDA-1
          (make-lam (list 'num1 'num2) (make-arith '* (list 'num1
'num2))))
        (define LAMBDA-2
          (make-lam (list 'num1 'num2) (make-arith '* (list 'num2
'num1))))
        (define EXPR-1
          (make-lam
           '(num1 num3)
           (make-lam
            '(num2 num4)
            (make-arith '* (list 'num1 'num2 (make-arith '* (list
'num3 'num4)))))))
```

```
        (define EXPR-2
          (make-lam
           '(num3 num1)
           (make-lam
            '(num4 num2)
            (make-arith '* (list 'num3 'num4 (make-arith '* (list
'num1 'num2)))))))
        (define SUBTRACT (make-lam (list 'num1 'num2) (make-arith '-
'(5 6))))
        (define ADD-FN
          (make-def 'addtwonums (list 'num1 'num2) (make-arith '+
(list 'num1 'num2))))
        (define WRONG-ARITH-EXPR (make-arith '+ (list #f)))
        (define WRONG-BOOL-EXPR (make-bool 'and (list 1 #f)))
        (define WRONG-IF-EXPR (make-if-exp (make-bool 'and (list #t
#t)) '- #f))
        (define PROGRAM-WITH-ERR-EXPRS-ONLY
          (list WRONG-ARITH-EXPR WRONG-BOOL-EXPR WRONG-IF-EXPR))
        (define MK-ADD-DEF
          (make-def 'mk-add '(n) (make-lam '(m) (make-arith '+ '(n
m)))))
        (define ADD50-DEF
          (make-def 'add50 '(x) (make-call (make-call 'mk-add '(50))
'(x))))
        (define ADD50-OR-6-DEF
          (make-def
           'add50-or-6
           '(y)
           (make-if-exp
            (make-cmp '< '(y 0))
            (make-call 'add50 '(y))
            (make-call (make-call 'mk-add '(6)) '(y)))))
        (define (world-after-drag world start-x start-y stop-x stop-y)
          (let* ((world1 (handle-mouse world start-x start-y "button-
down"))
                 (world2 (handle-mouse world1 start-x start-y "drag"))
                 (world3 (handle-mouse world2 stop-x stop-y "drag"))
                 (world4 (handle-mouse world3 stop-x stop-y "button-
up")))
            world4))
        (define (drag-shape s start-x start-y stop-x stop-y)
          (send (send (send (send s handle-mouse start-x start-y
"button-down") handle-mouse start-x start-y "drag") handle-mouse stop-
x stop-y "drag") handle-mouse
            stop-x
            stop-y
            "button-up"))
        (define (world-after-point-click world click-x click-y)
          (let* ((world1 (handle-mouse world click-x click-y "button-
down"))
```

```
                 (world2 (handle-mouse world1 click-x click-y "button-
up")))
            world2))
        (define (world-select-pointer world) (world-after-point-click
world 5 5))
        (define (world-select-rectangle world) (world-after-point-
click world 5 25))
        (define (world-select-circle world) (world-after-point-click
world 5 45))
        (define (world-select-triangle world) (world-after-point-click
world 5 65))
        (define (apply-one-action action world)
          (let* ((action-verb (first action)) (action-params (rest
action)))
            (cond
             ((string=? action-verb "select-pointer") (world-select-
pointer world))
             ((string=? action-verb "select-rectangle") (world-select-
rectangle world))
             ((string=? action-verb "select-circle") (world-select-
circle world))
             ((string=? action-verb "select-triangle") (world-select-
triangle world))
             ((string=? action-verb "drag")
              (apply world-after-drag world action-params))
             (else (error "unknown action verb" action-verb)))))
        (define (drive-world world script) (foldl apply-one-action
world script))
        (define (get-world-shape-bounds world)
          (map (lambda (sh) (send sh get-bounds)) (get-world-shapes
world)))
        (define (bounds-after-clicks w script)
          (get-world-shape-bounds (drive-world w script)))
        (define RECT-START1 '(45 50))
        (define RECT-FINISH1 '(108 205))
        (define RECT1-BBOX (append RECT-START1 RECT-FINISH1))

Test Case:
  (test-equal?
   "basic ops"
   (eval
    (append ARITHMETIC-EXPR-PROGRAM COMPARISON-PROGRAM BOOLEAN-EXPR-
PROGRAM))
   (append (list 246 0 110.25 5) (list #t #t #t) (list #t #t #f #f)))
Test Result: Success

Test Case:
  (test-equal? "if" (eval IF-EXPR-PROGRAM) (list 35 0))
Test Result: Success
Test Case:
  (test-equal?
   "call-fn evaluates to a function"
   (eval
    (list
     MK-ADD-DEF
     ADD50-DEF
     ADD50-OR-6-DEF
     (make-call 'add50 '(100))
     (make-call 'add50-or-6 '(2))
     (make-call 'add50-or-6 '(-2))))
   (list 150 8 48))
Test Result: Failure
actual : Error: Failed to define `(eval p)`, because: second: list
contains too few elements
  list: (list (def 'add50-or-6 '(y) (object:If-Exp% ...)))
expected : (150 8 48)
expression : (check-equal? (eval (list MK-ADD-DEF ADD50-DEF ADD50-
OR-6-DEF (make-call (quote add50) (quote (100))) (make-call (quote
add50-or-6) (quote (2))) (make-call (quote add50-or-6) (quote (-2)))))
(list 150 8 48))
params : (Error: Failed to define `(eval p)`, because: second: list
contains too few elements
  list: (list (def 'add50-or-6 '(y) (object:If-Exp% ...))) (150 8 48))

Test Case:
  (test-equal? "fn call before def" (eval (list CALL-SQR-DEF SQR-DEF))
(list 4))
Test Result: Failure
actual : Error: Failed to define `(eval p)`, because: second: list
contains too few elements
  list: (list (def 'Square '(num1) (object:Arith% ...)))
expected : (4)
expression : (check-equal? (eval (list CALL-SQR-DEF SQR-DEF)) (list
4))
params : (Error: Failed to define `(eval p)`, because: second: list
contains too few elements
  list: (list (def 'Square '(num1) (object:Arith% ...))) (4))

Test Case:
  (test-= "fn call" (first (eval CIRCUMFERENCE-OF-CIRCLE-PROG)) 31.42
0.001)
Test Result: Error
first: contract violation
  expected: (and/c list? (not/c empty?))
  given: "Error: Failed to define `(eval p)`, because: symbol=?:
contract violation\n  expected: symbol?\n  given: (object:Var% ...)\n
argument position: 1st\n  other arguments...:\n   'radius"

Test Case:
  (test-equal?
   "fn call returns err"
   (eval PROGRAM-CHK-DIVIDE-BY-ZERO)
   (list 2 "Cannot Divide by Zero"))
Test Result: Failure
actual : Error: Failed to define `(eval p)`, because: second: list
contains too few elements
  list: (list (def 'avoidDivideByZero '(num1 num2) (object:If-Exp
% ...)))
expected : (2 Cannot Divide by Zero)
expression : (check-equal? (eval PROGRAM-CHK-DIVIDE-BY-ZERO) (list 2
Cannot Divide by Zero))
params : (Error: Failed to define `(eval p)`, because: second: list
contains too few elements
  list: (list (def 'avoidDivideByZero '(num1 num2) (object:If-Exp
% ...))) (2 Cannot Divide by Zero))

Test Case:
  (test-equal? "larger program" (eval FINAL-PROGRAM) (list 246 110.25
4 #t 365))
Test Result: Failure
actual : Error: Failed to define `(eval p)`, because: second: list
contains too few elements
  list: (list (def 'Square '(num1) (object:Arith% ...)))
expected : (246 110.25 4 #t 365)
expression : (check-equal? (eval FINAL-PROGRAM) (list 246 110.25 4 #t
365))
params : (Error: Failed to define `(eval p)`, because: second: list
contains too few elements
  list: (list (def 'Square '(num1) (object:Arith% ...))) (246 110.25 4
#t 365))

Test Case:
  (test-equal?
   "basic subst test1"
   (subst 4 'num1 (make-arith '+ (list 3 'num1)))
   (make-arith '+ (list 3 4)))
Test Result: Failure
actual : #(struct:arith + (#(struct:object:Number% ...)
#(struct:object:Number% ...)))
expected : #(struct:arith + (3 4))
expression : (check-equal? (subst 4 (quote num1) (make-arith (quote +)
(list 3 (quote num1)))) (make-arith (quote +) (list 3 4)))
params : (#(struct:arith + (#(struct:object:Number% ...)
#(struct:object:Number% ...))) #(struct:arith + (3 4)))

Test Case:
  (test-equal?
   "subst unused var"
   (subst 5 'num1 (make-arith '+ (list 5 'num)))
   (make-arith '+ (list 5 'num)))
Test Result: Failure
actual : #(struct:arith + (#(struct:object:Number% ...)
#(struct:object:Var% ...)))
expected : #(struct:arith + (5 num))
expression : (check-equal? (subst 5 (quote num1) (make-arith (quote +)
(list 5 (quote num)))) (make-arith (quote +) (list 5 (quote num))))
params : (#(struct:arith + (#(struct:object:Number% ...)
#(struct:object:Var% ...))) #(struct:arith + (5 num)))

Test Case:
  (test-equal?
   "subst shadowed var"
   (subst 10 'y (make-lam '(x y) (make-arith '* '(y x))))
   (make-lam '(x y) (make-arith '* '(y x))))
Test Result: Failure
actual : #(struct:lam (#(struct:object:Var% ...) #(struct:object:Var
% ...)) #(struct:object:Arith% ...))
expected : #(struct:lam (x y) #(struct:arith * (y x)))
expression : (check-equal? (subst 10 (quote y) (make-lam (quote (x y))
(make-arith (quote *) (quote (y x))))) (make-lam (quote (x y)) (make-
arith (quote *) (quote (y x)))))
params : (#(struct:lam (#(struct:object:Var% ...) #(struct:object:Var
% ...)) #(struct:object:Arith% ...)) #(struct:lam (x y) #(struct:arith
* (y x))))

Test Case:
  (test-equal?
   "subst nested lam"
   (subst
    (make-lam '(x) (make-arith '+ '(x 5)))
    'y
    (make-lam '(a) (make-call 'y (list (make-lam '(b c) 'y)))))
   (make-lam
    '(a)
    (make-call
     (make-lam '(x) (make-arith '+ '(x 5)))
     (list (make-lam '(b c) (make-lam '(x) (make-arith '+ '(x
5))))))))
Test Result: Failure
actual : #(struct:lam (#(struct:object:Var% ...)) #(struct:object:Call
% ...))
expected : #(struct:lam (a) #(struct:call #(struct:lam (x)
#(struct:arith + (x 5))) (#(struct:lam (b c) #(struct:lam (x)
#(struct:arith + (x 5)))))))
expression : (check-equal? (subst (make-lam (quote (x)) (make-arith
(quote +) (quote (x 5)))) (quote y) (make-lam (quote (a)) (make-call
(quote y) (list (make-lam (quote (b c)) (quote y)))))) (make-lam
(quote (a)) (make-call (make-lam (quote (x)) (make-arith (quote +)
(quote (x 5)))) (list (make-lam (quote (b c)) (make-lam (quote (x))
(make-arith (quote +) (quote (x 5)))))))))
params : (#(struct:lam (#(struct:object:Var% ...))
#(struct:object:Call% ...)) #(struct:lam (a) #(struct:call
#(struct:lam (x) #(struct:arith + (x 5))) (#(struct:lam (b c)
#(struct:lam (x) #(struct:arith + (x 5))))))))

Test Case:
  (test-equal?
   "static distance: one lam"
   (expr->expr/no-var TEST-EXPR-FOR-EXPR-WITH-NO-VAR)
   TEST-EXPR-WITH-NO-VAR-RESULT)
Test Result: Failure
actual : #(struct:lam/no-var #(struct:arith + ((0 0) (1 0))))
expected : #(struct:lam/no-var #(struct:arith + ((0 0) (0 1))))
expression : (check-equal? (expr->expr/no-var TEST-EXPR-FOR-EXPR-WITH-
NO-VAR) TEST-EXPR-WITH-NO-VAR-RESULT)
params : (#(struct:lam/no-var #(struct:arith + ((0 0) (1 0))))
#(struct:lam/no-var #(struct:arith + ((0 0) (0 1)))))

Test Case:
  (test-equal?
   "static distance: nested lambdas"
   (expr->expr/no-var (make-lam '(x y) (make-lam '(z) (make-call 'x
'(y z)))))
   (make-lam/no-var (make-lam/no-var (make-call '(1 0) '((1 1) (0
0))))))
Test Result: Failure
actual : #(struct:lam/no-var #(struct:lam/no-var #(struct:call (1 0)
((2 0) (0 0)))))
expected : #(struct:lam/no-var #(struct:lam/no-var #(struct:call (1 0)
((1 1) (0 0)))))
expression : (check-equal? (expr->expr/no-var (make-lam (quote (x y))
(make-lam (quote (z)) (make-call (quote x) (quote (y z)))))) (make-
lam/no-var (make-lam/no-var (make-call (quote (1 0)) (quote ((1 1) (0
0)))))))
params : (#(struct:lam/no-var #(struct:lam/no-var #(struct:call (1 0)
((2 0) (0 0))))) #(struct:lam/no-var #(struct:lam/no-var #(struct:call
(1 0) ((1 1) (0 0))))))

Test Case:
  (test-equal?
   "static distance: shadowed vars"
   (expr->expr/no-var
    (make-lam '(x y) (make-lam '(y x) (make-arith '/ '(x x x y)))))
   (make-lam/no-var
    (make-lam/no-var (make-arith '/ '((0 1) (0 1) (0 1) (0 0))))))
Test Result: Failure
actual : #(struct:lam/no-var #(struct:lam/no-var #(struct:arith / ((1
0) (1 0) (1 0) (0 0)))))
expected : #(struct:lam/no-var #(struct:lam/no-var #(struct:arith /
((0 1) (0 1) (0 1) (0 0)))))
expression : (check-equal? (expr->expr/no-var (make-lam (quote (x y))
(make-lam (quote (y x)) (make-arith (quote /) (quote (x x x y))))))
(make-lam/no-var (make-lam/no-var (make-arith (quote /) (quote ((0 1)
(0 1) (0 1) (0 0)))))))
params : (#(struct:lam/no-var #(struct:lam/no-var #(struct:arith / ((1
0) (1 0) (1 0) (0 0))))) #(struct:lam/no-var #(struct:lam/no-var
#(struct:arith / ((0 1) (0 1) (0 1) (0 0))))))

Test Case:
  (test-equal?
   "static distance: same vars"
   (expr->expr/no-var
    (make-lam '(x) (make-call (make-lam '(x) (make-call 'x '(x)))
'(x))))
   (make-lam/no-var
    (make-call (make-lam/no-var (make-call '(0 0) '((0 0)))) '((0
0)))))
Test Result: Success

Test Case:
  (test-equal? "not expr=?" (expr=? LAMBDA-1 LAMBDA-2) #f)
Test Result: Success

Test Case:
  (test-equal? "expr=" (expr=? EXPR-1 EXPR-2) #t)
Test Result: Success

Test Case:
  (test-true
   "lambda evaluates to itself but dont eval body"
   (let ((res (first (eval (list SUBTRACT)))))
     (and (lam? res) (equal? res SUBTRACT))))
Test Result: Failure
expression : (check-true (let ((res (first (eval (list SUBTRACT)))))
(and (lam? res) (equal? res SUBTRACT))))
params : (#f)

Test Case:
  (test-pred
   "evaluate to lambda"
   lam?
   (first
    (eval
     (list
      (make-call
       (make-call (make-lam '(x) (make-call 'x '(x))) (list (make-lam
'(x) 'x)))
       (list (make-lam '(x) 'x)))))))
Test Result: Error
```

```
first: contract violation
  expected: (and/c list? (not/c empty?))
  given: "Error: Failed to define `(eval p)`, because: symbol=?:
contract violation\n  expected: symbol?\n  given: (object:Var% ...)\n
argument position: 1st\n  other arguments...:\n   'x"

Test Case:
  (test-equal? "errs" (andmap string? (eval PROGRAM-WITH-ERR-EXPRS-
ONLY)) #t)
Test Result: Success

Test Case:
  (test-set=?
   "Create one rectangle"
   (bounds-after-clicks
    INITIAL-WORLD
    `(("select-rectangle") ("drag" ,@RECT-START1 ,@RECT-FINISH1)))
   (list RECT1-BBOX))
Test Result: Success

Test Case:
  (test-check
   "Create one triangle"
   bounds-set-approx=?
   (bounds-after-clicks
    INITIAL-WORLD
    `(("select-triangle") ("drag" 80 80 120 80)))
   (list (list 60 45 120 115)))
Test Result: Success

Test Case:
  (test-check
   "Create two triangles"
   bounds-set-approx=?
   (bounds-after-clicks
    INITIAL-WORLD
    `(("select-triangle") ("drag" 80 80 120 80) ("drag" 120 80 120
68)))
   (list (list 60 45 120 115) (list 110 68 130 86)))
Test Result: Success

Test Case:
  (test-check
   "Create two triangles; drag with a resize and a move"
   bounds-set-approx=?
   (bounds-after-clicks
    INITIAL-WORLD
    `(("select-triangle")
      ("drag" 80 80 120 80)
      ("drag" 120 80 120 68)
      ("select-pointer")
      ("drag" 120 80 100 80)))
   (list (list 70 63 100 97) (list 90 68 110 86)))
Test Result: Success

Test Case:
  (test-check
   "Create a triangle and a rectangle; resize both"
   bounds-set-approx=?
   (bounds-after-clicks
    INITIAL-WORLD
    `(("select-triangle")
      ("drag" 80 80 120 80)
      ("select-rectangle")
      ("drag" 120 80 135 42)
      ("select-pointer")
      ("drag" 120 80 80 60)))
   (list (list 63 60 97 90) (list 80 42 135 60)))
Test Result: Success


Results for Suite draw+eval-tests:
  Test Successes: 11
  Test Failures: 12
  Test Errors: 2

Raw Score: 11/25
Normalized Score: 4/10


Overall Results:
  Test Successes: 11
  Test Failures: 12
  Test Errors: 2

Raw Score: 11/25
Normalized Score: 4/10
```