

```

1. ;;PROC
2. ; let f = proc(x) (x - 11) in (f (f 77))
3. (define p8
4.   (a-program (let-exp 'f (proc-exp 'x (diff-exp (var-exp 'x) (const-exp 11)))
5.               (call-exp (var-exp 'f) (call-exp (var-exp 'f) (const-exp 77))))))
6. ;; = 55
7. ; (proc (f) (f (f 77)))
8. ; proc (x) (x - 11) )
9. (define p9
10.  (a-program (call-exp (proc-exp 'f (call-exp (var-exp 'f) (call-exp (var-exp 'f) (const-exp
11.  77))))
12.              (proc-exp 'x (diff-exp (var-exp 'x) (const-exp 11))))))
13. ;; = 55
14. ; let x = 200
15. ; in let f = proc(z) (z - x)
16. ;   in let x = 100
17. ;     in let g = proc (z) (z - x)
18. ;       in (f 1) - (g 1)
19. (define p10
20.  (a-program (let-exp 'x (const-exp 200)
21.                  (let-exp 'f (proc-exp 'z (diff-exp (var-exp 'z) (var-exp 'x)))
22.                      (let-exp 'x (const-exp 100)
23.                          (let-exp 'g (proc-exp 'z (diff-exp (var-exp 'z)
24.  (var-exp 'x)))
25.                              (diff-exp (call-exp (var-exp 'f) (const-exp
26.  1)))
27.                                  (call-exp (var-exp 'g) (const-exp
28.  1))))))))))
29. ;; = -100
30. ; let f = proc(w)
31. ;   let x = 100
32. ;   in proc (z) (z - x)
33. ; in let x = 200 in ((f 0) 0)
34. (define p11
35.  (a-program
36.   (let-exp 'f
37.    (proc-exp 'w
38.     (let-exp 'x
39.      (const-exp 100)
40.      (proc-exp 'z (diff-exp (var-exp 'z) (var-exp 'x)) )))
41.   (let-exp 'x
42.    (const-exp 200)
43.    (call-exp (call-exp (var-exp 'f) (const-exp 0)) (const-exp 0))))))
44. ;; = -100
45. ;let f = proc(x) 10 in
46. ; let f = proc (x) if zero? x then 1 else (f (x - 1))
47. ;   in (f 3)
48. (define p12
49.  (a-program
50.   (let-exp 'f
51.    (proc-exp 'x
52.     (const-exp 10))
53.   (let-exp 'f
54.    (proc-exp 'x
55.     (if-exp (zero?-exp (var-exp 'x))
56.              (const-exp 1)
57.              (call-exp (var-exp 'f) (diff-exp (var-exp 'x) (const-exp
58.  1))))))
59.   (call-exp (var-exp 'f) (const-exp 3))
60.   )))
61. ;; = 10
62. ; let x=3 in let y=4 in (let x = y-5 in x-y) - x
63. (define p13
64.  (a-program
65.   (let-exp 'x
66.    (const-exp 3)
67.   (let-exp 'y
68.    (const-exp 4)
69.   (diff-exp

```

```

65.         (let-exp 'x
66.             (diff-exp (var-exp 'y) (const-exp 5))
67.             (diff-exp (var-exp 'x) (var-exp 'y)))
68.         (var-exp 'x))))))
69. ;; = -8
70. ; Infinite loop: if you interpret the program below, the interpreter will
71. ; go in an infinite loop. Interrupt it with the Stop button in the top-right corner
72. (define p14
73.   (a-program
74.     (call-exp
75.       (proc-exp 'x
76.         (call-exp (var-exp 'x) (var-exp 'x)))
77.       (proc-exp 'x
78.         (call-exp (var-exp 'x) (var-exp 'x)))
79.     )))
80.
81.
82. ;; LETREC
83. ; letrec double(x) =
84. ;   if zero?(x) then 0 else (double (x - 1)) - (-2)
85. ; in (double 6)
86. (define p11
87.   (a-program (letrec-exp 'double 'x
88.                         (if-exp (zero?-exp (var-exp 'x))
89.                                   (const-exp 0)
90.                                   (diff-exp (call-exp (var-exp 'double) (diff-exp (var-exp
91. 'x) (const-exp 1))))
92.                                   (const-exp -2))))
93.   (call-exp (var-exp 'double) (const-exp 6))))
94. ;; = 12
95. ;let f = proc(x) 10 in
96. ; letrec f(x) = if zero? x then 1 else (f (x - 1))
97. ; in (f 3)
98. (define p12
99.   (a-program
100.    (let-exp 'f
101.      (proc-exp 'x
102.        (const-exp 10))
103.      (letrec-exp 'f
104.        'x
105.        (if-exp (zero?-exp (var-exp 'x))
106.                  (const-exp 1)
107.                  (call-exp (var-exp 'f) (diff-exp (var-exp 'x) (const-exp 1))))
108.        (call-exp (var-exp 'f) (const-exp 3))
109.      )))
110. ;; = 1

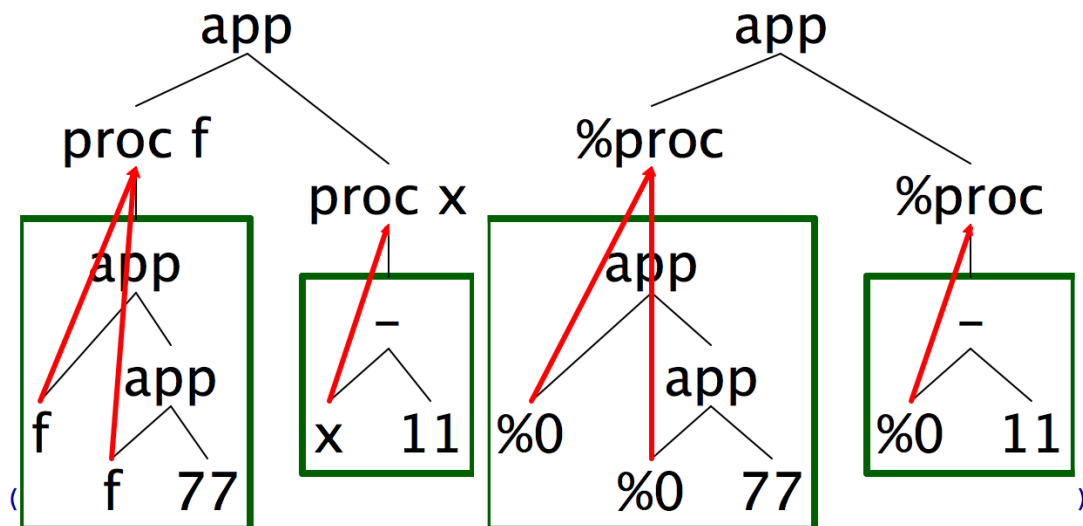
```

CONTOUR DIAGRAMS

```

;( proc (f) (f (f 77))
; proc (x) (x - 11) )
(define p9
  (a-program (call-exp (proc-exp 'f (call-exp (var-exp 'f) (call-exp (var-exp 'f)
    (const-exp 77))))
    (proc-exp 'x (diff-exp (var-exp 'x) (const-exp 11))))))

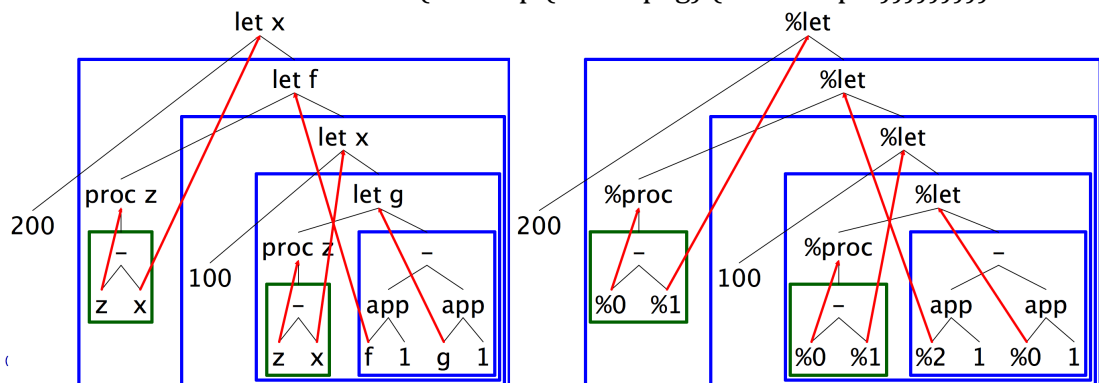
```



```

; let x = 200
; in let f = proc(z) (z - x)
;   in let x = 100
;     in let g = proc (z) (z - x)
;       in (f 1) - (g 1)
(define p10
  (a-program (let-exp 'x (const-exp 200)
    (let-exp 'f (proc-exp 'z (diff-exp (var-exp 'z) (var-exp 'x)))
      (let-exp 'x (const-exp 100)
        (let-exp 'g (proc-exp 'z (diff-exp (var-exp 'z) (var-exp 'x)))
          (diff-exp (call-exp (var-exp 'f) (const-exp 1))
            (call-exp (var-exp 'g) (const-exp 1))))))))))

```



```

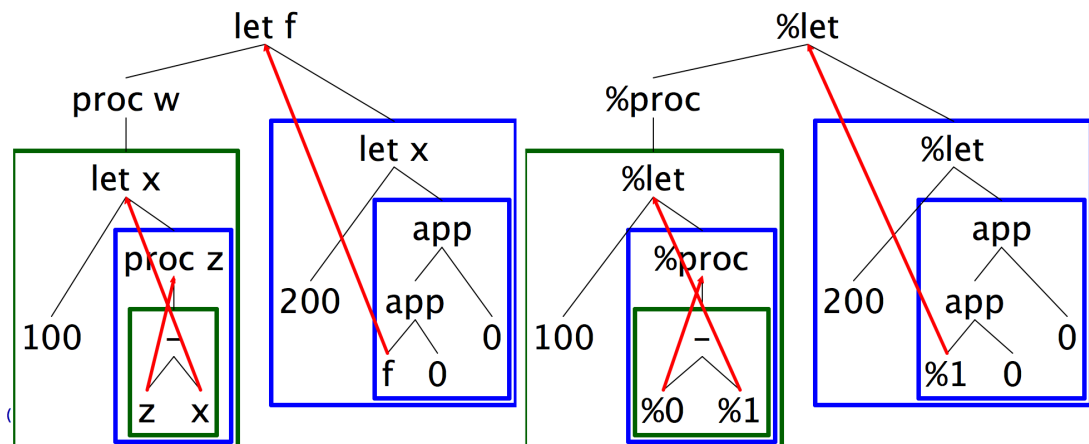
; let f = proc(w)
;   let x = 100
;   in proc (z) (z - x)
; in let x = 200 in ((f 0) 0)

```

```

(define p11
  (a-program
    (let-exp 'f
      (proc-exp 'w
        (let-exp 'x
          (const-exp 100)
          (proc-exp 'z (diff-exp (var-exp 'z) (var-exp 'x)) )))
      (let-exp 'x
        (const-exp 200)
        (call-exp (call-exp (var-exp 'f) (const-exp 0)) (const-exp 0))))))

```



```

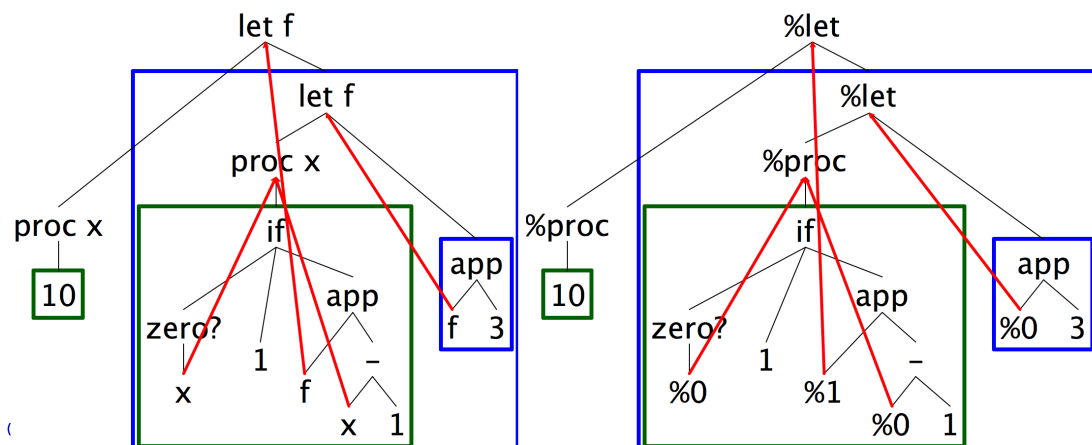
;let f = proc(x) 10 in
; let f = proc (x) if zero? x then 1 else (f (x - 1))
;   in (f 3)

```

```

(define p12
  (a-program
    (let-exp 'f
      (proc-exp 'x
        (const-exp 10))
      (let-exp 'f
        (proc-exp 'x
          (if-exp (zero?-exp (var-exp 'x))
            (const-exp 1)
            (call-exp (var-exp 'f) (diff-exp (var-exp 'x) (const-exp 1)))))
        (call-exp (var-exp 'f) (const-exp 3))
        ))))

```

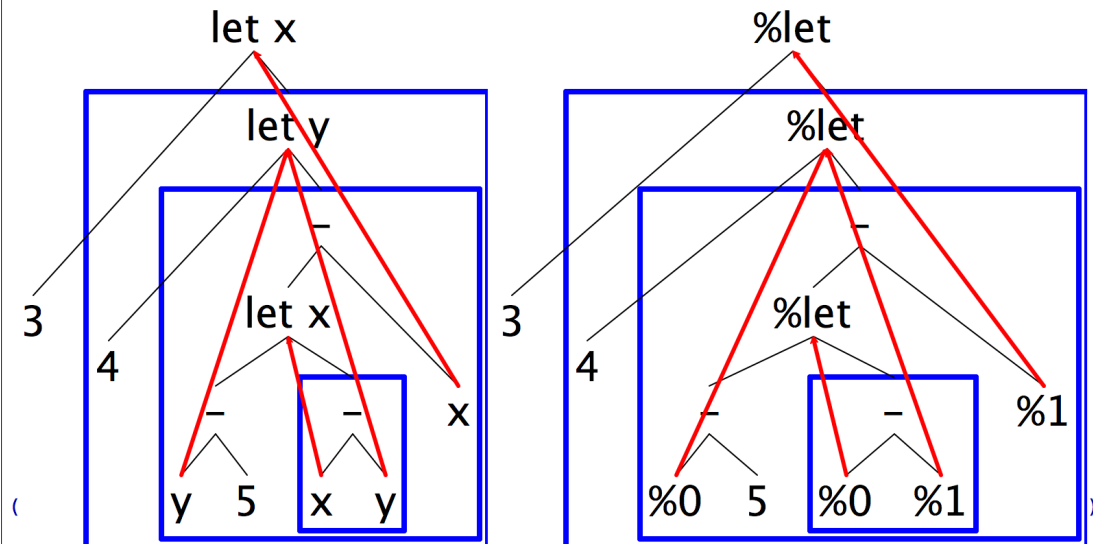


• let x=3 in let v=4 in (let x = v-5 in x-v) - x

```

(define p13
  (a-program
    (let-exp 'x
      (const-exp 3)
      (let-exp 'y
        (const-exp 4)
        (diff-exp
          (let-exp 'x
            (diff-exp (var-exp 'y) (const-exp 5))
            (diff-exp (var-exp 'x) (var-exp 'y)))
          (var-exp 'x))))))

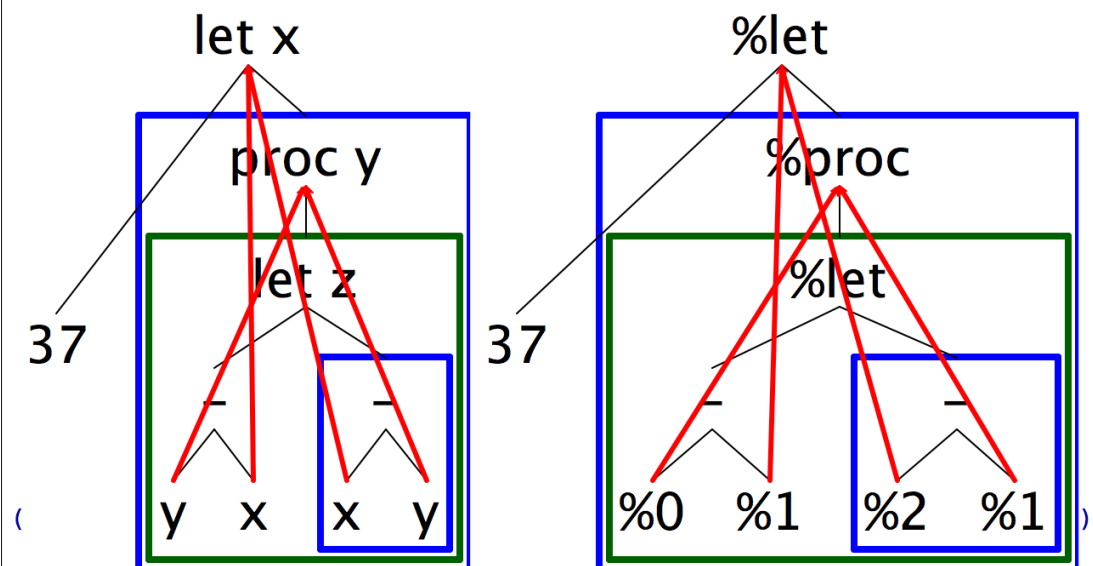
```



```

; let x= 37
; in proc (y)
;   let z = y - x
;   in x - y
(define p14
  (a-program
    (let-exp 'x
      (const-exp 37)
      (proc-exp 'y
        (let-exp 'z
          (diff-exp (var-exp 'y) (var-exp 'x))
          (diff-exp (var-exp 'x) (var-exp 'y))))))

```



```

1. ;;EXPLICIT-REFS
2. ; let g =
3. ;   let counter = newref(0)
4. ;   in proc (dummy)
5. ;     begin
6. ;       setref(counter, -(deref(counter), -1));
7. ;       deref(counter)
8. ;     end
9. ; in let a = (g 11) in let b = (g 11) in -(a,b)
10. (define p16
11.   (a-program (let-exp 'g
12.                 (let-exp 'counter
13.                   (newref-exp (const-exp 0))
14.                   (proc-exp 'dummy
15.                     (begin-exp
16.                       (setref-exp (var-exp 'counter) (diff-exp (deref-
17.                         exp (var-exp 'counter)) (const-exp -1))))
18.                       (list (deref-exp (var-exp 'counter))))))
19.                 (let-exp 'a
20.                   (call-exp (var-exp 'g) (const-exp 11))
21.                   (let-exp 'b
22.                     (call-exp (var-exp 'g) (const-exp 11))
23.                     (diff-exp (var-exp 'a) (var-exp 'b)))))))
24. ;; RESULT:
25. ;;#(struct:num-val -1)
26. ;;STORE:
27. ;;((0 #(struct:num-val 2)))
28. ; store a proc-val
29. ; newref(proc(dummy) 10)
30. (define p17
31.   (a-program (newref-exp (proc-exp 'dummy
32.                                 (const-exp 10))))))
33. ;;RESULT:
34. ;;#(struct:ref-val 0)
35. ;;STORE:
36. ;;((0 #(struct:proc-val #(struct:procedure dummy #(struct:const-exp 10) #(struct:extend-env
37.   i #(struct:num-val 1) #(struct:extend-env v #(struct:num-val 5) #(struct:extend-env x #
38.   (struct:num-val 10) #(struct:empty-env)))))))
39. ; create a chain of 2 references
40. ; newref (newref (newref(0)))
41. (define p20
42.   (a-program
43.     (newref-exp (newref-exp (newref-exp (const-exp 0))))))
44. ;;RESULT:
45. ;;#(struct:ref-val 2)
46. ;;STORE:
47. ;;((0 #(struct:num-val 0)) (1 #(struct:ref-val 0)) (2 #(struct:ref-val 1)))
48. ; create a cycle of memory refs of length 10
49. ; let b = newref(0) in
50. ; let e = newref(b) in
51. ; letrec chain(n) =
52. ;   if iszero? n
53. ;   then 0
54. ;   else
55. ;     let x = newref(0) in begin;
56. ;       setref(deref(e),x);
57. ;       setref(e,x);
58. ;       chain(n-1)
59. ;     end
60. ; in begin chain(10); setref(deref(e),b) end
61. (define p21
62.   (a-program
63.     (let-exp 'b
64.       (newref-exp (const-exp 0))
65.       (let-exp 'e

```

```

67.      (newref-exp (var-exp 'b))
68.      (letrec-exp '(chain) '(n)
69.        (list (if-exp (zero?-exp (var-exp 'n))
70.                  (const-exp 0)
71.                  (let-exp 'x
72.                        (newref-exp (const-exp 0))
73.                        (begin-exp
74.                          (setref-exp (deref-exp (var-exp 'e)) (var-exp
75.                            'x))
76.                          (list (setref-exp (var-exp 'e) (var-exp 'x))
77.                                (call-exp (var-exp 'chain) (diff-exp (var-
78.                                  exp 'n) (const-exp 1))))))
79.                        ))
80.                        (begin-exp
81.                          (call-exp (var-exp 'chain) (const-exp 10))
82.                          (list (setref-exp (deref-exp (var-exp 'e)) (var-
83.                            exp 'b))
84.                                ))
85.                        ))))
86.      ;;RESULT:
87.      ;;#(struct:num-val 23)
88.      ;;STORE:
89.      ;;((0 #(struct:ref-val 2)) (1 #(struct:ref-val 11)) (2 #(struct:ref-val 3)) (3 #(struct:ref-
90.        val 4)) (4 #(struct:ref-val 5)) (5 #(struct:ref-val 6)) (6 #(struct:ref-val 7)) (7 #
91.        (struct:ref-val 8)) (8 #(struct:ref-val 9)) (9 #(struct:ref-val 10)) (10 #(struct:ref-val
92.        11)) (11 #(struct:ref-val 0)))

```

```

1. ;; IMPLICIT-REFS
2. ; Let x = 0 in Let x = 2 in x
3. (define p7
4.   (a-program (let-exp 'x (const-exp 0)
5.                 (let-exp 'x (const-exp 2)
6.                   (var-exp 'x))))))
7. ;;STORE:
8. ;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10)) (3 #(struct:num-
9.   val 0)) (4 #(struct:num-val 2)))
10. ; Let f = proc(x) (x - 11) in (f (f 7))
11. (define p8
12.   (a-program (let-exp 'f (proc-exp 'x (diff-exp (var-exp 'x) (const-exp 11)))
13.             (call-exp (var-exp 'f) (call-exp (var-exp 'f) (const-exp 7))))))
14.
15. ;;STORE:
16. ;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10))
17. ;; (3 #(struct:proc-val #(struct:procedure x #(struct:diff-exp #(struct:var-exp x) #
18.   (struct:const-exp 11))
19. ;; #(struct:extend-env i 0 #(struct:extend-env v 1 #(struct:extend-env x 2 #(struct:empty-
20.   env))))))
21. ;; (4 #(struct:num-val 7)) (5 #(struct:num-val -4)))
22. ; Let x = 200
23. ; in Let f = proc(z) (z - x)
24. ;   in Let x = 100
25. ;     in Let g = proc (z) (z - x)
26. ;       in (f 1) - (g 1)
27. (define p10
28.   (a-program (let-exp 'x (const-exp 200)
29.                 (let-exp 'f (proc-exp 'z (diff-exp (var-exp 'z) (var-exp 'x)))
30.                   (let-exp 'x (const-exp 100)
31.                     (let-exp 'g (proc-exp 'z (diff-exp (var-exp 'z)
32.   (var-exp 'x)))
33.   (diff-exp (call-exp (var-exp 'f) (const-exp
34.   1)))
35.   (call-exp (var-exp 'g) (const-exp
36.   1)))))))))
37. ;;STORE:
38. ;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10)) (3 #(struct:num-
39.   val 200))
40. ;; (4 #(struct:proc-val #(struct:procedure z #(struct:diff-exp #(struct:var-exp z) #
41.   (struct:var-exp x)) #(struct:extend-env x 3 #(struct:extend-env i 0 #(struct:extend-env v 1
42.   #(struct:extend-env x 2 #(struct:empty-env))))))
43. ;; (5 #(struct:num-val 100)) (6 #(struct:proc-val #(struct:procedure z #(struct:diff-exp #
44.   (struct:var-exp z) #(struct:var-exp x)) #(struct:extend-env x 5 #(struct:extend-env f 4 #
45.   (struct:extend-env x 3 #(struct:extend-env i 0 #(struct:extend-env v 1 #(struct:extend-env x
46.   2 #(struct:empty-env))))))
47. ;; (7 #(struct:num-val 1)) (8 #(struct:num-val 1)))
48. ; Let g =
49. ;   Let counter = 0
50. ;   in proc (dummy)
51. ;     begin
52. ;       counter := counter - -1;
53. ;       counter
54. ;     end
55. ; in Let a = (g 11) in Let b = (g 11) in a - b
56. (define p16
57.   (a-program (let-exp 'g
58.                 (let-exp 'counter
59.                   (const-exp 0)
60.                   (proc-exp 'dummy
61.                     (begin-exp
62.   (assign-exp 'counter (diff-exp (var-exp 'counter)
63.   (const-exp -1)))
64.   (list (var-exp 'counter))))))
65.   (let-exp 'a

```



```

57.         (call-exp (var-exp 'g) (const-exp 11))
58.         (let-exp 'b
59.             (call-exp (var-exp 'g) (const-exp 11))
60.             (diff-exp (var-exp 'a) (var-exp 'b))))))
61.
62. ;;STORE:
63. ;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10))
64. ;;(3 #(struct:num-val 2)) (4 #(struct:proc-val #(struct:procedure dummy #(struct:begin-exp #
65. (struct:assign-exp counter #(struct:diff-exp #(struct:var-exp counter) #(struct:const-exp
66. -1))) (#(struct:var-exp counter))) #(struct:extend-env counter 3 #(struct:extend-env i 0 #
67. (struct:extend-env v 1 #(struct:extend-env x 2 #(struct:empty-env)))))) (5 #(struct:num-
68. val 11)) (6 #(struct:num-val 1)) (7 #(struct:num-val 11)) (8 #(struct:num-val 2)))
69.
70. ; illustrate lack of referential transparency:
71. ; let g =
72. ;   let counter = 0
73. ;   in proc (dummy)
74. ;     begin
75. ;       counter := counter - 1;
76. ;       counter
77. ;     end
78. ; in (g 11) - (g 11)
79. (define p17
80.   (a-program (let-exp 'g
81.                     (let-exp 'counter
82.                         (const-exp 0)
83.                         (proc-exp 'dummy
84.                             (begin-exp
85.                                 (assign-exp 'counter (diff-exp (var-exp 'counter)
86.                                                                    (list (var-exp 'counter))))
87.                                 (diff-exp (call-exp (var-exp 'g) (const-exp 11)) (call-exp (var-exp
88. 'g) (const-exp 11)))))))
89.
90. ;;STORE:
91. ;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10)) (3 #(struct:num-
92. val 2)) (4 #(struct:proc-val #(struct:procedure dummy #(struct:begin-exp #(struct:assign-exp
93. counter #(struct:diff-exp #(struct:var-exp counter) #(struct:const-exp -1))) (#(struct:var-
94. exp counter))) #(struct:extend-env counter 3 #(struct:extend-env i 0 #(struct:extend-env v 1
95. #(struct:extend-env x 2 #(struct:empty-env)))))) (5 #(struct:num-val 11)) (6 #(struct:num-
96. val 11)))

```

```

1. ;; LETREC-CONT
2. ; Let x = 0 in x - 4
3. (define p6
4.   (a-program (let-exp 'x
5.                 (const-exp 0)
6.                 (diff-exp (var-exp 'x) (const-exp 4))))))
7. ;Evaluating Let x = 0 in (x - 4) in cc [ ]
8. ;Evaluating 0 in cc Let x = [ ] in <(x - 4)>
9. ;Sending Number: 0 to cc Let x = [ ] in <(x - 4)>
10. ;Evaluating (x - 4) in cc [ ]
11. ;Evaluating x in cc ([ ] - <4>)
12. ;Sending Number: 0 to cc ([ ] - <4>)
13. ;Evaluating 4 in cc (0 - [ ])
14. ;Sending Number: 4 to cc (0 - [ ])
15. ;Sending Number: -4 to cc [ ]
16. ;End of computation.
17.
18. ; Let f = proc(x) (x - 11) in (f (f 7))
19. (define p8
20.   (a-program (let-exp 'f (proc-exp 'x (diff-exp (var-exp 'x) (const-exp 11)))
21.               (call-exp (var-exp 'f) (call-exp (var-exp 'f) (const-exp 7))))))
22. ;Evaluating Let f = proc(x)(x - 11) in (f (f 7)) in cc [ ]
23. ;Evaluating proc(x)(x - 11) in cc Let f = [ ] in <(f (f 7))>
24. ;Sending Procedure: {proc} to cc Let f = [ ] in <(f (f 7))>
25. ;Evaluating (f (f 7)) in cc [ ]
26. ;Evaluating f in cc ([ ] <(f 7)>)
27. ;Sending Procedure: {proc} to cc ([ ] <(f 7)>)
28. ;Evaluating (f 7) in cc ({proc} [ ])
29. ;Evaluating f in cc ({proc} ([ ] <7>))
30. ;Sending Procedure: {proc} to cc ({proc} ([ ] <7>))
31. ;Evaluating 7 in cc ({proc} ({proc} [ ])
32. ;Sending Number: 7 to cc ({proc} ({proc} [ ])
33. ;Evaluating (x - 11) in cc ({proc} [ ])
34. ;Evaluating x in cc ({proc} ([ ] - <11>))
35. ;Sending Number: 7 to cc ({proc} ([ ] - <11>))
36. ;Evaluating 11 in cc ({proc} (7 - [ ])
37. ;Sending Number: 11 to cc ({proc} (7 - [ ])
38. ;Sending Number: -4 to cc ({proc} [ ])
39. ;Evaluating (x - 11) in cc [ ]
40. ;Evaluating x in cc ([ ] - <11>))
41. ;Sending Number: -4 to cc ([ ] - <11>))
42. ;Evaluating 11 in cc (-4 - [ ])
43. ;Sending Number: 11 to cc (-4 - [ ])
44. ;Sending Number: -15 to cc [ ]
45. ;End of computation.
46.
47. ;( proc (f) (f (f 77))
48. ; proc (x) (x - 11) )
49. (define p9
50.   (a-program (call-exp (proc-exp 'f (call-exp (var-exp 'f) (call-exp (var-exp 'f) (const-exp
51.   (proc-exp 'x (diff-exp (var-exp 'x) (const-exp 11))))))
52. ;Evaluating (proc(f)(f (f 77)) proc(x)(x - 11)) in cc [ ]
53. ;Evaluating proc(f)(f (f 77)) in cc ([ ] <proc(x)(x - 11)>))
54. ;Sending Procedure: {proc} to cc ([ ] <proc(x)(x - 11)>))
55. ;Evaluating proc(x)(x - 11) in cc ({proc} [ ])
56. ;Sending Procedure: {proc} to cc ({proc} [ ])
57. ;Evaluating (f (f 77)) in cc [ ]
58. ;Evaluating f in cc ([ ] <(f 77)>))
59. ;Sending Procedure: {proc} to cc ([ ] <(f 77)>))
60. ;Evaluating (f 77) in cc ({proc} [ ])
61. ;Evaluating f in cc ({proc} ([ ] <77>))
62. ;Sending Procedure: {proc} to cc ({proc} ([ ] <77>))
63. ;Evaluating 77 in cc ({proc} ({proc} [ ])
64. ;Sending Number: 77 to cc ({proc} ({proc} [ ])
65. ;Evaluating (x - 11) in cc ({proc} [ ])
66. ;Evaluating x in cc ({proc} ([ ] - <11>))
67. ;Sending Number: 77 to cc ({proc} ([ ] - <11>))
68. ;Evaluating 11 in cc ({proc} (77 - [ ])

```

```

69. ;Sending Number: 11 to cc ({proc} (77 - [ ]))
70. ;Sending Number: 66 to cc ({proc} [ ])
71. ;Evaluating (x - 11) in cc [ ]
72. ;Evaluating x in cc ([ ] - <11>)
73. ;Sending Number: 66 to cc ([ ] - <11>)
74. ;Evaluating 11 in cc (66 - [ ])
75. ;Sending Number: 11 to cc (66 - [ ])
76. ;Sending Number: 55 to cc [ ]
77. ;End of computation.
78.
79. ; (if zero? 0 then 1 else 2) - (let x= 5 in x)
80. (define p13
81.   (a-program (diff-exp (if-exp (zero?-exp (const-exp 0))
82.                                (const-exp 1)
83.                                (const-exp 2))
84.               (let-exp 'x
85.                       (const-exp 5)
86.                       (var-exp 'x))
87.               )))
88. ;Evaluating (if zero? 0 then 1 else 2 - let x = 5 in x) in cc [ ]
89. ;Evaluating if zero? 0 then 1 else 2 in cc ([ ] - <let x = 5 in x>)
90. ;Evaluating zero? 0 in cc (if [ ] then <1> else <2> - <let x = 5 in x>)
91. ;Evaluating 0 in cc (if zero?([ ]) then <1> else <2> - <let x = 5 in x>)
92. ;Sending Number: 0 to cc (if zero?([ ]) then <1> else <2> - <let x = 5 in x>)
93. ;Sending Boolean: #t to cc (if [ ] then <1> else <2> - <let x = 5 in x>)
94. ;Evaluating 1 in cc ([ ] - <let x = 5 in x>)
95. ;Sending Number: 1 to cc ([ ] - <let x = 5 in x>)
96. ;Evaluating let x = 5 in x in cc (1 - [ ])
97. ;Evaluating 5 in cc (1 - let x = [ ] in <x>)
98. ;Sending Number: 5 to cc (1 - let x = [ ] in <x>)
99. ;Evaluating x in cc (1 - [ ])
100. ;Sending Number: 5 to cc (1 - [ ])
101. ;Sending Number: -4 to cc [ ]
102. ;End of computation.
103.
104. ; Let f = proc(x) 7 in f(2) - f(3)
105. (define p14
106.   (a-program (let-exp 'f (proc-exp 'x (const-exp 7))
107.               (diff-exp (call-exp (var-exp 'f) (const-exp 2))
108.                           (call-exp (var-exp 'f) (const-exp 3))))))
109. ;Evaluating Let f = proc(x)7 in ((f 2) - (f 3)) in cc [ ]
110. ;Evaluating proc(x)7 in cc let f = [ ] in <((f 2) - (f 3))>
111. ;Sending Procedure: {proc} to cc let f = [ ] in <((f 2) - (f 3))>
112. ;Evaluating ((f 2) - (f 3)) in cc [ ]
113. ;Evaluating (f 2) in cc ([ ] - <(f 3)>)
114. ;Evaluating f in cc (([ ] <2>) - <(f 3)>)
115. ;Sending Procedure: {proc} to cc (([ ] <2>) - <(f 3)>)
116. ;Evaluating 2 in cc (({proc} [ ] - <(f 3)>)
117. ;Sending Number: 2 to cc (({proc} [ ] - <(f 3)>)
118. ;Evaluating 7 in cc ([ ] - <(f 3)>)
119. ;Sending Number: 7 to cc ([ ] - <(f 3)>)
120. ;Evaluating (f 3) in cc (7 - [ ])
121. ;Evaluating f in cc (7 - ([ ] <3>))
122. ;Sending Procedure: {proc} to cc (7 - ([ ] <3>))
123. ;Evaluating 3 in cc (7 - ({proc} [ ]))
124. ;Sending Number: 3 to cc (7 - ({proc} [ ]))
125. ;Evaluating 7 in cc (7 - [ ])
126. ;Sending Number: 7 to cc (7 - [ ])
127. ;Sending Number: 0 to cc [ ]
128. ;End of computation.

```