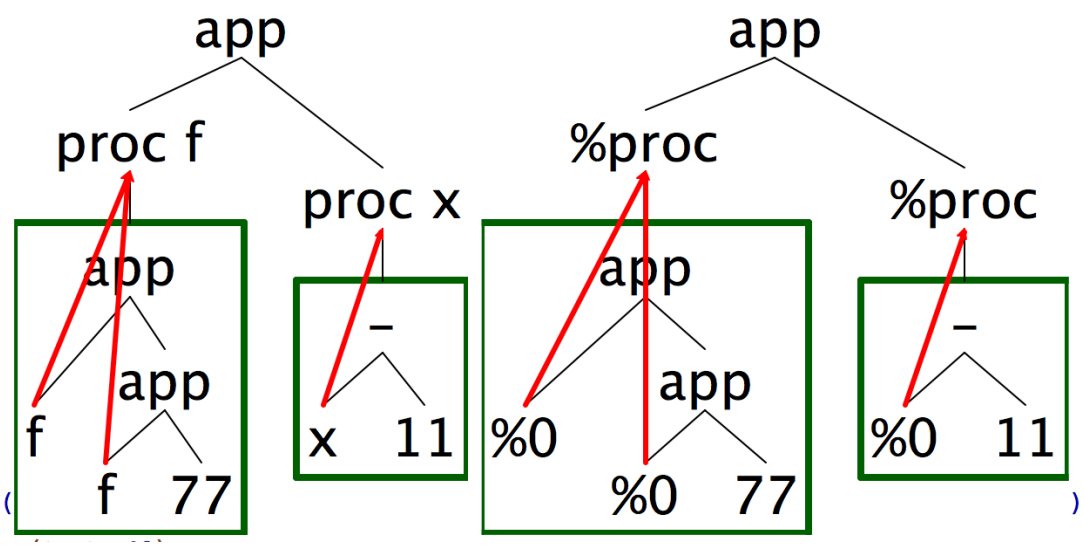
```
;;PROC
; let f = proc(x) (x - 11) in (f (f 77))
(define p8
  (a-program (let-exp 'f (proc-exp 'x (diff-exp (var-exp 'x) (const-exp 11)))
                      (call-exp (var-exp 'f) (call-exp (var-exp 'f) (const-exp 77))))))
;; = 55
;( proc (f) (f (f 77))
;  proc (x) (x - 11) )
(define p9
  (a-program (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))))))
;; = 55
; 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)))))))))
;; = -100
; 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))))))
;; = -100
;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))
                     ))))
;; = 10
; let x=3 in let y=4 in (let x = y-5 in x-y) - x
(define p13
  (a-program
   (let-exp 'x
            (const-exp 3)
            (let-exp 'y
                     (const-exp 4)
                     (diff-exp
```

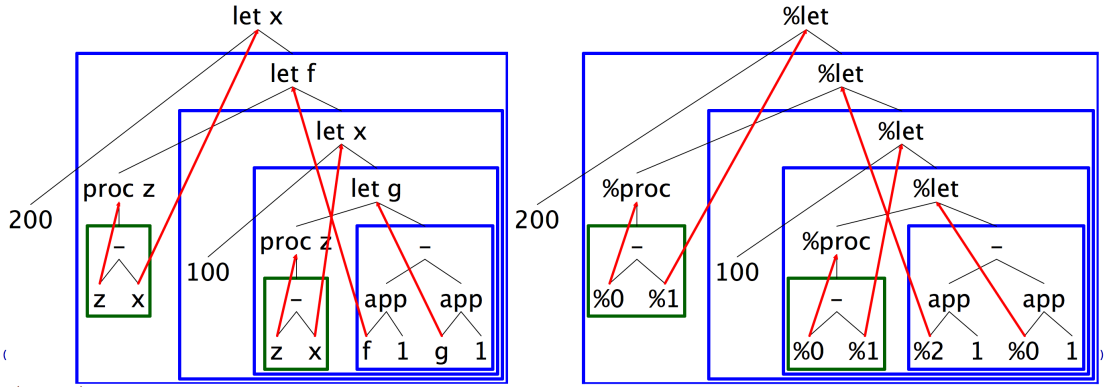
```
                    (let-exp 'x
                             (diff-exp (var-exp 'y) (const-exp 5))
                             (diff-exp (var-exp 'x) (var-exp 'y)))
                    (var-exp 'x))))))
;; = -8
; Infinite loop: if you interpret the program below, the interpreter will
; go in an infinite loop. Interrupt it with the Stop button in the top-right corner
(define p14
  (a-program
   (call-exp
    (proc-exp 'x
              (call-exp (var-exp 'x) (var-exp 'x)))
    (proc-exp 'x
              (call-exp (var-exp 'x) (var-exp 'x)))
    )))


;;LETREC
; letrec double(x) =
;   if zero?(x) then 0 else (double (x - 1)) - (-2)
; in (double 6)
(define p11
  (a-program (letrec-exp 'double 'x
                         (if-exp (zero?-exp (var-exp 'x))
                                 (const-exp 0)
                                 (diff-exp (call-exp (var-exp 'double) (diff-exp (var-exp 'x) (const-exp 1)))
                                           (const-exp -2)))
                         (call-exp (var-exp 'double) (const-exp 6)))))
;; = 12
;let f = proc(x) 10 in
;  letrec f(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))
            (letrec-exp 'f
                        '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))
                        ))))
;; = 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))))))
```

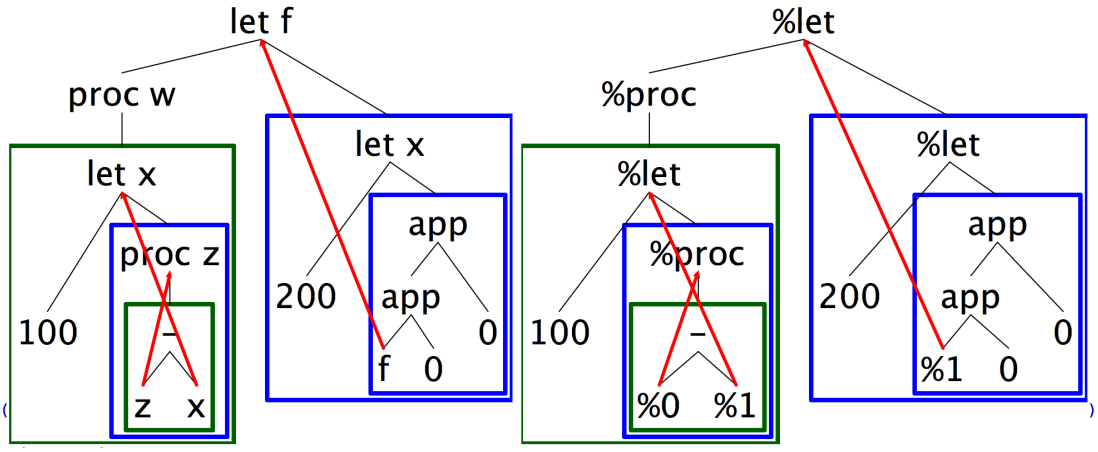



```
; let x = 200
; in let f = proc(z) (z - x)
;    in let x = 100
;       in let g = proc (z) (z - x)
;          in (f 1) - (g 1)
(define p10
  (a-program (let-exp 'x (const-exp 200)
                (let-exp 'f (proc-exp 'z (diff-exp (var-exp 'z) (var-exp 'x)))
                     (let-exp 'x (const-exp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

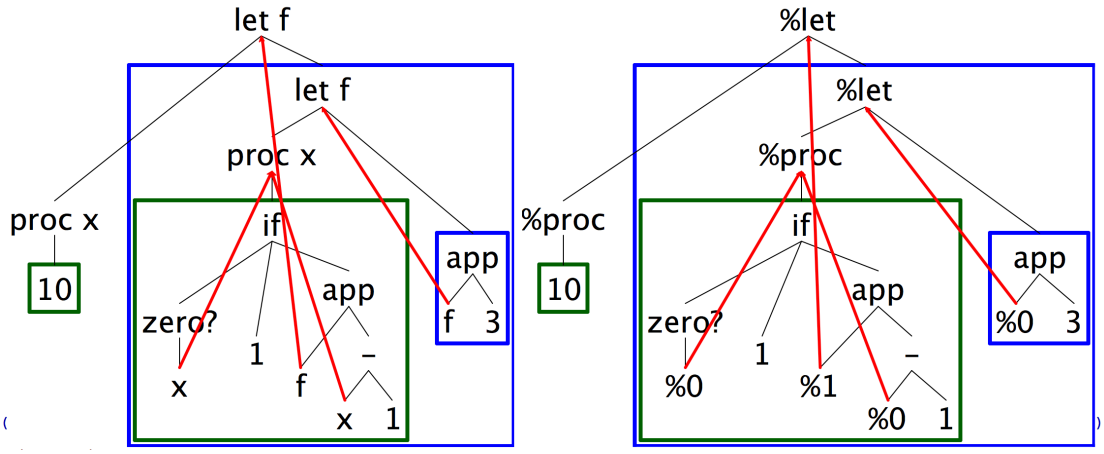



```
; let f = proc(w)
;          let x = 100
;          in proc (z) (z - x)
; in let x = 200 in ((f 0) 0)
```

```
(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))))))
```

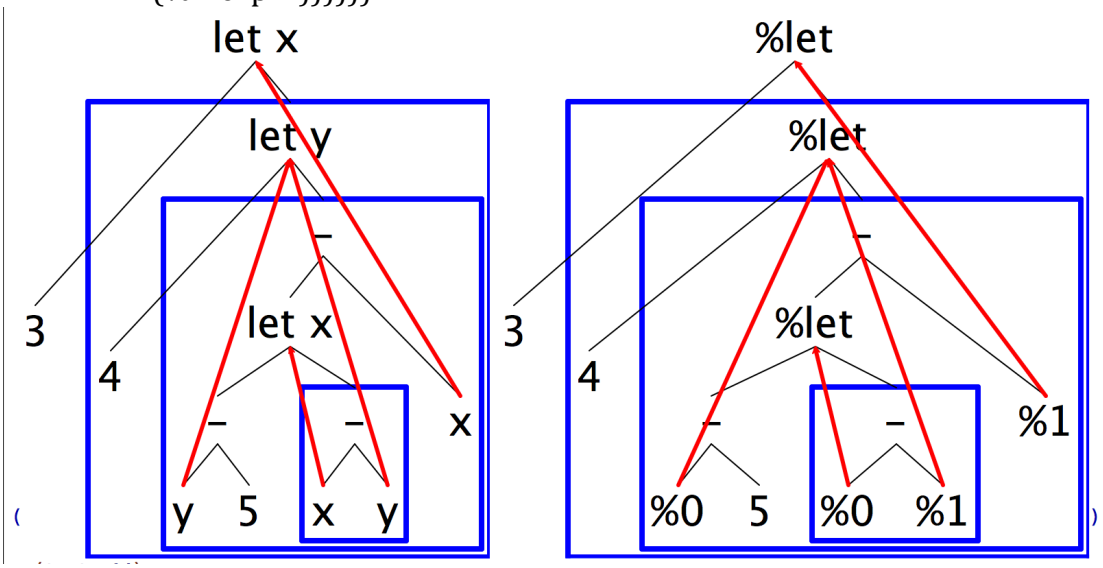



```
;let f = proc(x) 10 in
;  let f = proc (x) if zero? x then 1 else (f (x - 1))
;    in (f 3)
(define p12
 (a-program
  (let-exp 'f
        (proc-exp 'x
              (const-exp 10))
        (let-exp 'f
              (proc-exp 'x
               (if-exp (zero?-exp (var-exp 'x))
                    (const-exp 1)
                    (call-exp (var-exp 'f) (diff-exp (var-exp 'x) (const-exp 1)))))
              (call-exp (var-exp 'f) (const-exp 3))
              ))))
```

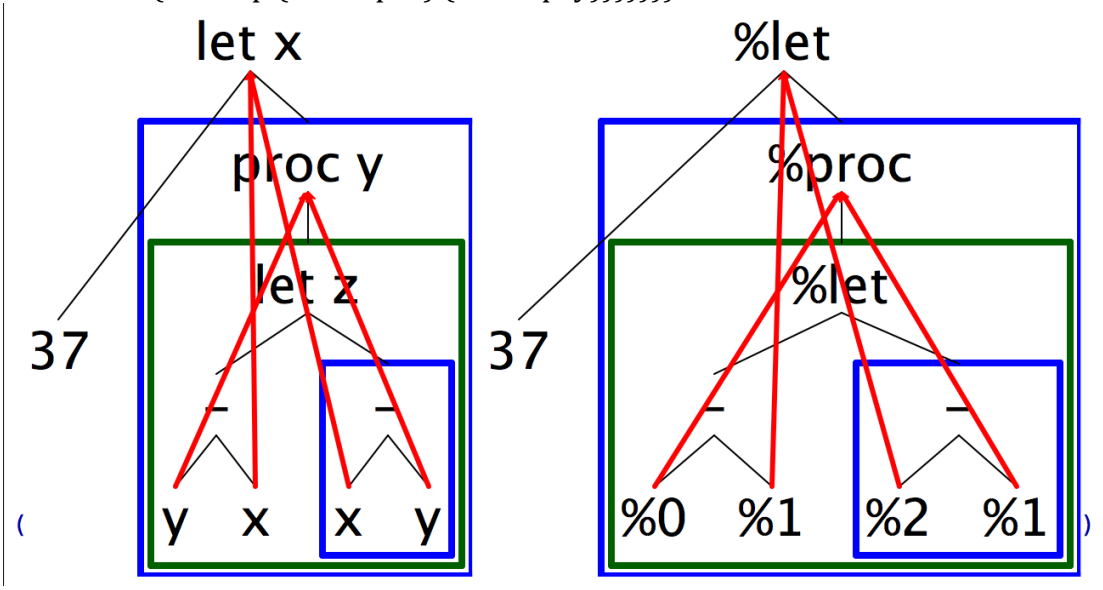


; let x=3 in let y=4 in (let x = y-5 in x-y) - x

```
(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)))))))
```



```
;;EXPLICIT-REFS
; let g =
;   let counter = newref(0)
;     in proc (dummy)
;         begin
;         setref(counter, -(deref(counter), -1));
;         deref(counter)
;         end
;  in let a = (g 11) in let b = (g 11) in -(a,b)
(define p16
  (a-program (let-exp 'g
                      (let-exp 'counter
                               (newref-exp (const-exp 0))
                               (proc-exp 'dummy
                                         (begin-exp
                                           (setref-exp (var-exp 'counter) (diff-exp (deref-exp (var-exp 'counter)) (const-exp -1)))
                                           (list (deref-exp (var-exp 'counter))))))
                      (let-exp 'a
                               (call-exp (var-exp 'g) (const-exp 11))
                               (let-exp 'b
                                        (call-exp (var-exp 'g) (const-exp 11))
                                        (diff-exp (var-exp 'a) (var-exp 'b)))))))
;; RESULT:
;;#(struct:num-val -1)
;;STORE:
;;((0 #(struct:num-val 2)))

; store a proc-val
; newref(proc(dummy) 10)
(define p17
  (a-program (newref-exp (proc-exp 'dummy
                                   (const-exp 10)))))
;;RESULT:
;;#(struct:ref-val 0)
;;STORE:
;;((0 #(struct:proc-val #(struct:procedure dummy #(struct:const-exp 10) #(struct:extend-env i #(struct:num-val 1) #(struct:extend-env v #(struct:num-val 5) #(struct:extend-env x #(struct:num-val 10) #(struct:empty-env)))))))

; create a chain of 2 references
; newref (newref (newref(0)))
(define p20
  (a-program
   (newref-exp (newref-exp (newref-exp (const-exp 0))))))

;;RESULT:
;;#(struct:ref-val 2)
;;STORE:
;;((0 #(struct:num-val 0)) (1 #(struct:ref-val 0)) (2 #(struct:ref-val 1)))

; create a cycle of memory refs of length 10
; let b = newref(0) in
; let e = newref(b) in
; letrec chain(n) =
;   if iszero? n
;     then 0
;     else
;      let x = newref(0) in begin;
;                           setref(deref(e),x);
;                           setref(e,x);
;                           chain(n-1)
;                           end
; in begin chain(10); setref(deref(e),b) end
(define p21
  (a-program
   (let-exp 'b
            (newref-exp (const-exp 0))
            (let-exp 'e
```

```
                    (newref-exp (var-exp 'b))
                    (letrec-exp '(chain) '(n)
                        (list (if-exp (zero?-exp (var-exp 'n))
                                (const-exp 0)
                                (let-exp 'x
                                        (newref-exp (const-exp 0))
                                        (begin-exp
                                          (setref-exp (deref-exp (var-exp 'e)) (var-exp 'x))
                                          (list (setref-exp (var-exp 'e) (var-exp 'x))
                                                (call-exp (var-exp 'chain) (diff-exp (var-exp 'n) (const-exp 1))))))
                                ))
                        (begin-exp
                                          (call-exp (var-exp 'chain) (const-exp 10))
                                          (list (setref-exp (deref-exp (var-exp 'e)) (var-exp 'b))
                                                ))
                        )))))

;;RESULT:
;;#(struct:num-val 23)
;;STORE:
;;((0 #(struct:ref-val 2)) (1 #(struct:ref-val 11)) (2 #(struct:ref-val 3)) (3 #(struct:ref-val 4)) (4 #(struct:ref-val 5)) (5 #(struct:ref-val 6)) (6 #(struct:ref-val 7)) (7 #(struct:ref-val 8)) (8 #(struct:ref-val 9)) (9 #(struct:ref-val 10)) (10 #(struct:ref-val 11)) (11 #(struct:ref-val 0)))
```

```
;; IMPLICIT-REFS
; let x = 0 in let x = 2 in x
(define p7
  (a-program (let-exp 'x (const-exp 0)
                      (let-exp 'x (const-exp 2)
                               (var-exp 'x)))))
;;STORE:
;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10)) (3 #(struct:num-val 0)) (4 #(struct:num-val 2)))

; let f = proc(x) (x - 11) in (f (f 7))
(define p8
  (a-program (let-exp 'f (proc-exp 'x (diff-exp (var-exp 'x) (const-exp 11)))
                      (call-exp (var-exp 'f) (call-exp (var-exp 'f) (const-exp 7))))))

;;STORE:
;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10))
;; (3 #(struct:proc-val #(struct:procedure x #(struct:diff-exp #(struct:var-exp x) #(struct:const-exp 11))
;; #(struct:extend-env i 0 #(struct:extend-env v 1 #(struct:extend-env x 2 #(struct:empty-env)))))))
;; (4 #(struct:num-val 7)) (5 #(struct:num-val -4)))

; 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)))))))))

;;STORE:
;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10)) (3 #(struct:num-val 200))
;;(4 #(struct:proc-val #(struct:procedure z #(struct:diff-exp #(struct:var-exp z) #(struct:var-exp x)) #(struct:extend-env x 3 #(struct:extend-env i 0 #(struct:extend-env v 1 #(struct:extend-env x 2 #(struct:empty-env))))))))
;;(5 #(struct:num-val 100)) (6 #(struct:proc-val #(struct:procedure z #(struct:diff-exp #(struct:var-exp z) #(struct:var-exp x)) #(struct:extend-env x 5 #(struct:extend-env f 4 #(struct:extend-env x 3 #(struct:extend-env i 0 #(struct:extend-env v 1 #(struct:extend-env x 2 #(struct:empty-env)))))))))))
;;(7 #(struct:num-val 1)) (8 #(struct:num-val 1)))

; let g =
;   let counter = 0
;     in proc (dummy)
;         begin
;         counter := counter - -1;
;         counter
;         end
;  in let a = (g 11) in let b = (g 11) in a - b
(define p16
  (a-program (let-exp 'g
                      (let-exp 'counter
                               (const-exp 0)
                               (proc-exp 'dummy
                                         (begin-exp
                                           (assign-exp 'counter (diff-exp (var-exp 'counter) (const-exp -1)))
                                           (list (var-exp 'counter)))))
                      (let-exp 'a
```

```
                                (call-exp (var-exp 'g) (const-exp 11))
                                (let-exp 'b
                                         (call-exp (var-exp 'g) (const-exp 11))
                                         (diff-exp (var-exp 'a) (var-exp 'b)))))))

;;STORE:
;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10))
;;(3 #(struct:num-val 2)) (4 #(struct:proc-val #(struct:procedure dummy #(struct:begin-exp #
(struct:assign-exp counter #(struct:diff-exp #(struct:var-exp counter) #(struct:const-exp
-1))) (#(struct:var-exp counter))) #(struct:extend-env counter 3 #(struct:extend-env i 0 #
(struct:extend-env v 1 #(struct:extend-env x 2 #(struct:empty-env)))))))) (5 #(struct:num-
val 11)) (6 #(struct:num-val 1)) (7 #(struct:num-val 11)) (8 #(struct:num-val 2)))

; illustrate lack of referential transparency:
; let g =
;   let counter = 0
;     in proc (dummy)
;         begin
;         counter := counter - -1;
;         counter
;         end
;  in (g 11) - (g 11)
(define p17
  (a-program (let-exp 'g
                      (let-exp 'counter
                               (const-exp 0)
                               (proc-exp 'dummy
                                         (begin-exp
                                           (assign-exp 'counter (diff-exp (var-exp 'counter)
(const-exp -1)))
                                           (list (var-exp 'counter)))))
                      (diff-exp (call-exp (var-exp 'g) (const-exp 11)) (call-exp (var-exp
'g) (const-exp 11))))))

;;STORE:
;;((0 #(struct:num-val 1)) (1 #(struct:num-val 5)) (2 #(struct:num-val 10)) (3 #(struct:num-
val 2)) (4 #(struct:proc-val #(struct:procedure dummy #(struct:begin-exp #(struct:assign-exp
counter #(struct:diff-exp #(struct:var-exp counter) #(struct:const-exp -1))) (#(struct:var-
exp counter))) #(struct:extend-env counter 3 #(struct:extend-env i 0 #(struct:extend-env v 1
#(struct:extend-env x 2 #(struct:empty-env)))))))) (5 #(struct:num-val 11)) (6 #(struct:num-
val 11)))
```

```
;; LETREC-CONT
; let x = 0 in x - 4
(define p6
  (a-program (let-exp 'x
                                  (const-exp 0)
                                  (diff-exp (var-exp 'x) (const-exp 4)))))
;Evaluating let x = 0 in (x - 4) in cc [ ]
;Evaluating 0 in cc let x = [ ] in <(x - 4)>
;Sending Number: 0 to cc let x = [ ] in <(x - 4)>
;Evaluating (x - 4) in cc [ ]
;Evaluating x in cc ([ ] - <4>)
;Sending Number: 0 to cc ([ ] - <4>)
;Evaluating 4 in cc (0 - [ ])
;Sending Number: 4 to cc (0 - [ ])
;Sending Number: -4 to cc [ ]
;End of computation.

; let f = proc(x) (x - 11) in (f (f 7))
(define p8
  (a-program (let-exp 'f (proc-exp 'x (diff-exp (var-exp 'x) (const-exp 11)))
                      (call-exp (var-exp 'f) (call-exp (var-exp 'f) (const-exp 7))))))
;Evaluating let f = proc(x)(x - 11) in (f (f 7)) in cc [ ]
;Evaluating proc(x)(x - 11) in cc let f = [ ] in <(f (f 7))>
;Sending Procedure: {proc}  to cc let f = [ ] in <(f (f 7))>
;Evaluating (f (f 7)) in cc [ ]
;Evaluating f in cc ([ ] <(f 7)>)
;Sending Procedure: {proc}  to cc ([ ] <(f 7)>)
;Evaluating (f 7) in cc ({proc} [ ]
;Evaluating f in cc ({proc} ([ ] <7>)
;Sending Procedure: {proc}  to cc ({proc} ([ ] <7>)
;Evaluating 7 in cc ({proc} ({proc} [ ]
;Sending Number: 7 to cc ({proc} ({proc} [ ]
;Evaluating (x - 11) in cc ({proc} [ ]
;Evaluating x in cc ({proc} ([ ] - <11>)
;Sending Number: 7 to cc ({proc} ([ ] - <11>)
;Evaluating 11 in cc ({proc} (7 - [ ])
;Sending Number: 11 to cc ({proc} (7 - [ ])
;Sending Number: -4 to cc ({proc} [ ]
;Evaluating (x - 11) in cc [ ]
;Evaluating x in cc ([ ] - <11>)
;Sending Number: -4 to cc ([ ] - <11>)
;Evaluating 11 in cc (-4 - [ ])
;Sending Number: 11 to cc (-4 - [ ])
;Sending Number: -15 to cc [ ]
;End of computation.

;( 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))))))
;Evaluating (proc(f)(f (f 77)) proc(x)(x - 11)) in cc [ ]
;Evaluating proc(f)(f (f 77)) in cc ([ ] <proc(x)(x - 11)>)
;Sending Procedure: {proc}  to cc ([ ] <proc(x)(x - 11)>)
;Evaluating proc(x)(x - 11) in cc ({proc} [ ]
;Sending Procedure: {proc}  to cc ({proc} [ ]
;Evaluating (f (f 77)) in cc [ ]
;Evaluating f in cc ([ ] <(f 77)>)
;Sending Procedure: {proc}  to cc ([ ] <(f 77)>)
;Evaluating (f 77) in cc ({proc} [ ]
;Evaluating f in cc ({proc} ([ ] <77>)
;Sending Procedure: {proc}  to cc ({proc} ([ ] <77>)
;Evaluating 77 in cc ({proc} ({proc} [ ]
;Sending Number: 77 to cc ({proc} ({proc} [ ]
;Evaluating (x - 11) in cc ({proc} [ ]
;Evaluating x in cc ({proc} ([ ] - <11>)
;Sending Number: 77 to cc ({proc} ([ ] - <11>)
;Evaluating 11 in cc ({proc} (77 - [ ])
```

```
;Sending Number: 11 to cc ({proc} (77 - [ ])
;Sending Number: 66 to cc ({proc} [ ]
;Evaluating (x - 11) in cc [ ]
;Evaluating x in cc ([ ] - <11>)
;Sending Number: 66 to cc ([ ] - <11>)
;Evaluating 11 in cc (66 - [ ])
;Sending Number: 11 to cc (66 - [ ])
;Sending Number: 55 to cc [ ]
;End of computation.

; (if zero? 0 then 1 else 2) - (let x= 5 in x)
(define p13
  (a-program (diff-exp (if-exp (zero?-exp (const-exp 0))
                               (const-exp 1)
                               (const-exp 2))
                       (let-exp 'x
                                (const-exp 5)
                                (var-exp 'x))
                       )))
;Evaluating (if zero? 0 then 1 else 2 - let x = 5 in x) in cc [ ]
;Evaluating if zero? 0 then 1 else 2 in cc ([ ] - <let x = 5 in x>)
;Evaluating zero? 0 in cc (if [ ] then <1> else <2> - <let x = 5 in x>)
;Evaluating 0 in cc (if zero?([ ]) then <1> else <2> - <let x = 5 in x>)
;Sending Number: 0 to cc (if zero?([ ]) then <1> else <2> - <let x = 5 in x>)
;Sending Boolean: #t to cc (if [ ] then <1> else <2> - <let x = 5 in x>)
;Evaluating 1 in cc ([ ] - <let x = 5 in x>)
;Sending Number: 1 to cc ([ ] - <let x = 5 in x>)
;Evaluating let x = 5 in x in cc (1 - [ ])
;Evaluating 5 in cc (1 - let x = [ ] in <x>)
;Sending Number: 5 to cc (1 - let x = [ ] in <x>)
;Evaluating x in cc (1 - [ ])
;Sending Number: 5 to cc (1 - [ ])
;Sending Number: -4 to cc [ ]
;End of computation.

; let f = proc(x) 7 in f(2) - f(3)
(define p14
  (a-program (let-exp 'f (proc-exp 'x (const-exp 7))
                      (diff-exp (call-exp (var-exp 'f) (const-exp 2))
                                (call-exp (var-exp 'f) (const-exp 3))))))
;Evaluating let f = proc(x)7 in ((f 2) - (f 3)) in cc [ ]
;Evaluating proc(x)7 in cc let f = [ ] in <((f 2) - (f 3))>
;Sending Procedure: {proc}  to cc let f = [ ] in <((f 2) - (f 3))>
;Evaluating ((f 2) - (f 3)) in cc [ ]
;Evaluating (f 2) in cc ([ ] - <(f 3)>)
;Evaluating f in cc (([ ] <2>) - <(f 3)>)
;Sending Procedure: {proc}  to cc (([ ] <2>) - <(f 3)>)
;Evaluating 2 in cc (({proc} [ ] - <(f 3)>)
;Sending Number: 2 to cc (({proc} [ ] - <(f 3)>)
;Evaluating 7 in cc ([ ] - <(f 3)>)
;Sending Number: 7 to cc ([ ] - <(f 3)>)
;Evaluating (f 3) in cc (7 - [ ])
;Evaluating f in cc (7 - ([ ] <3>))
;Sending Procedure: {proc}  to cc (7 - ([ ] <3>))
;Evaluating 3 in cc (7 - ({proc} [ ])
;Sending Number: 3 to cc (7 - ({proc} [ ])
;Evaluating 7 in cc (7 - [ ])
;Sending Number: 7 to cc (7 - [ ])
;Sending Number: 0 to cc [ ]
;End of computation.
```