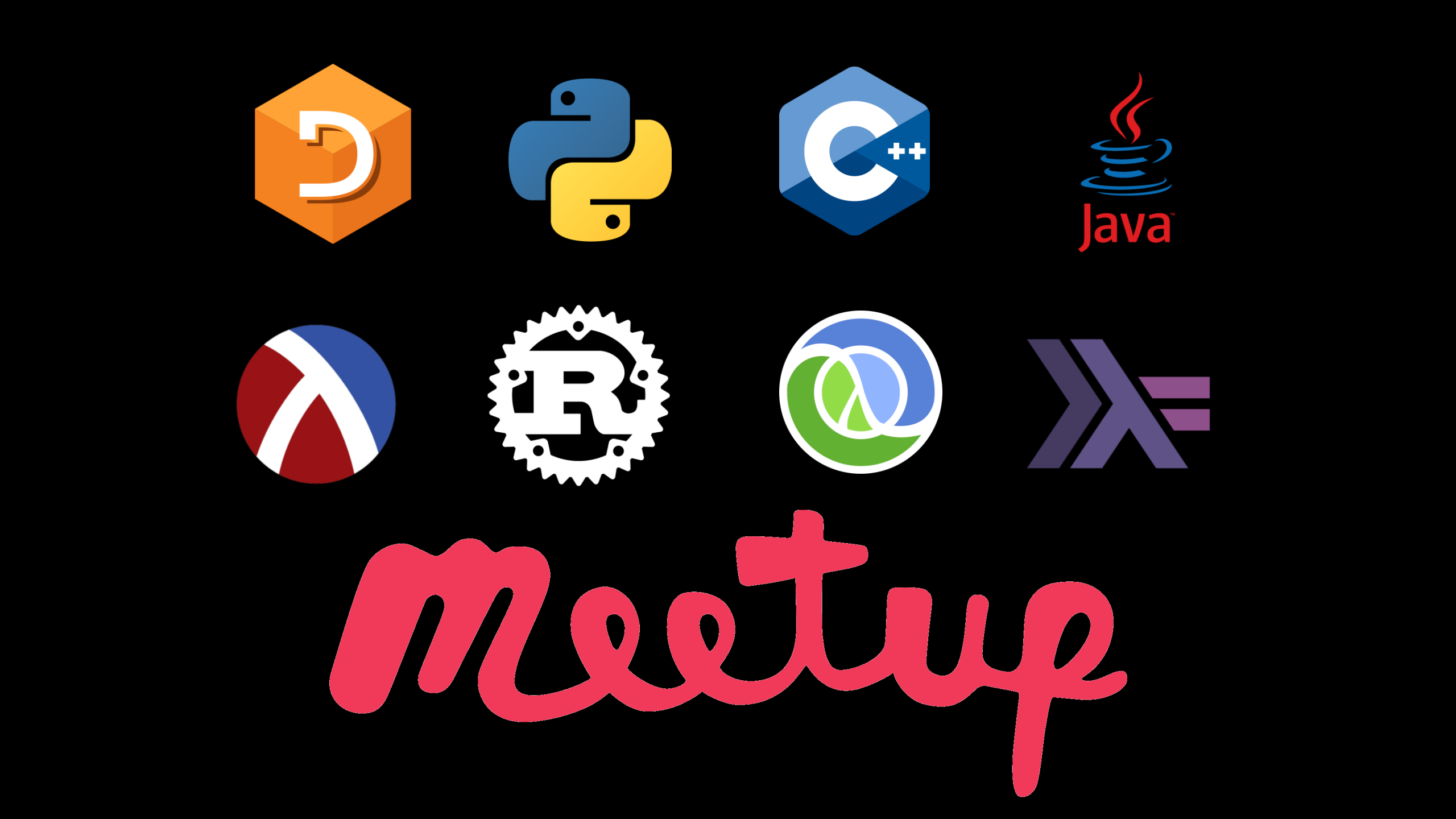
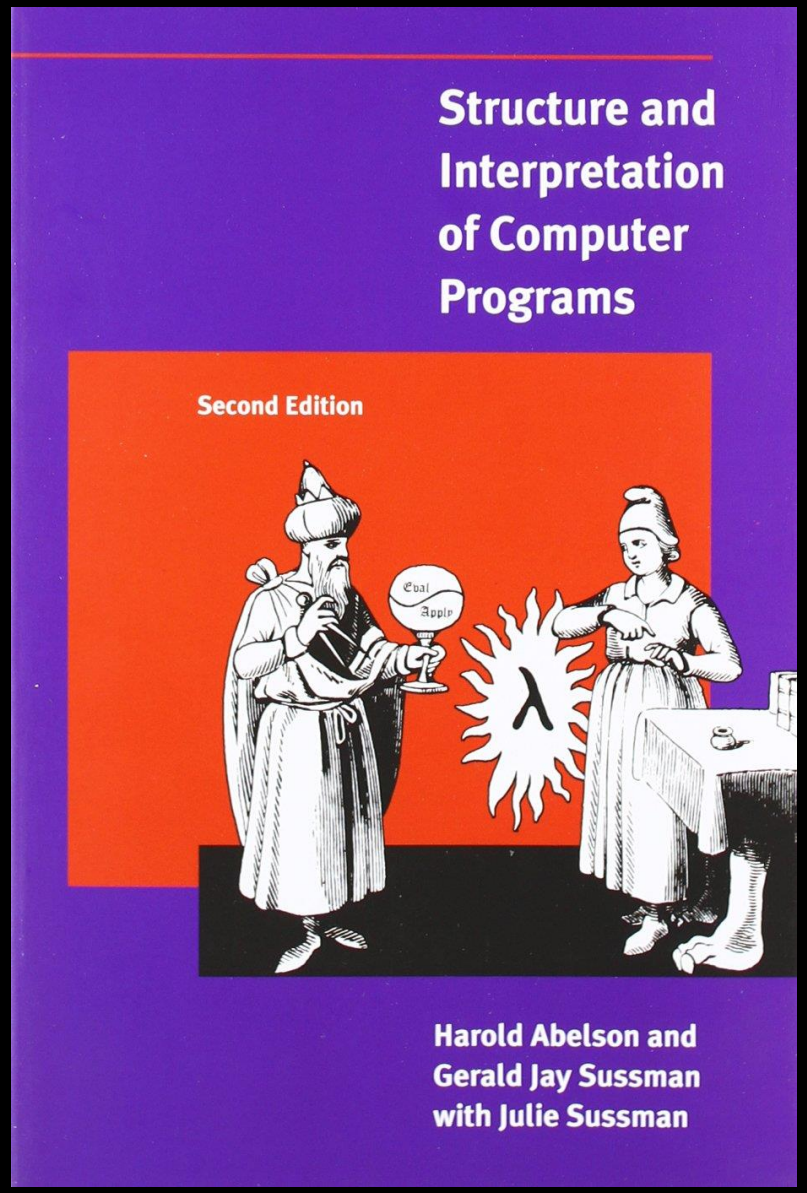

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 3.3

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

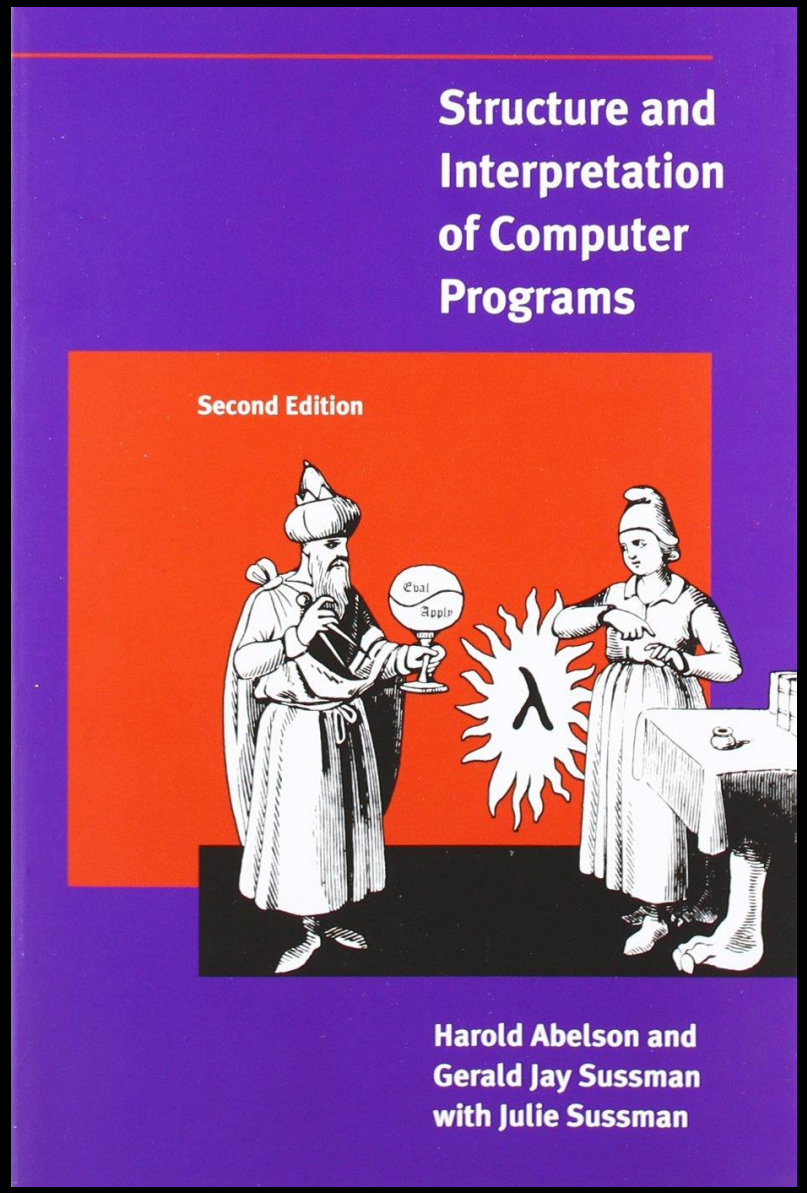
Following

Programming Languages Virtual Meetup

@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020



Structure and Interpretation of Computer Programs

Chaper 3.3

The primitive mutators for pairs are `set-car!` and `set-cdr!`. `set-car!` takes two arguments, the first of which must be a pair. It modifies this pair, replacing the `car` pointer by a pointer to the second argument of `set-car!`.[16]

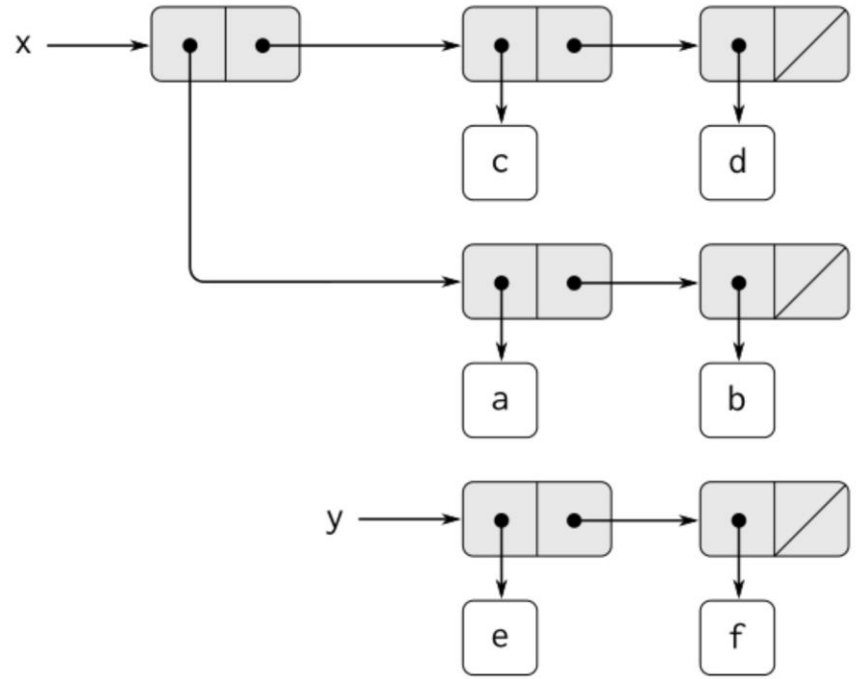


Figure 3.12: Lists x: `((a b) c d)` and y: `(e f)`.

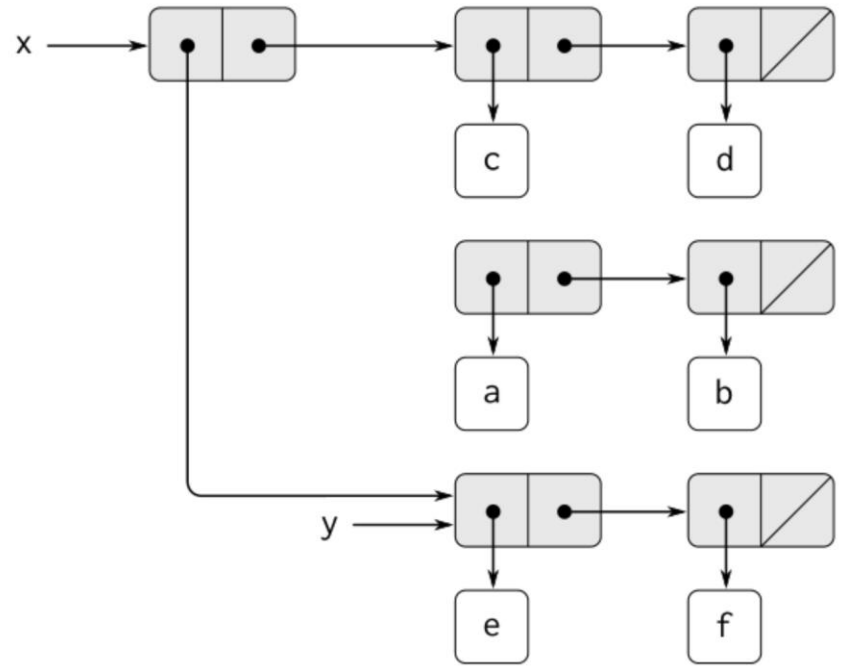


Figure 3.13: Effect of `(set-car! x y)` on the lists in Figure 3.12.

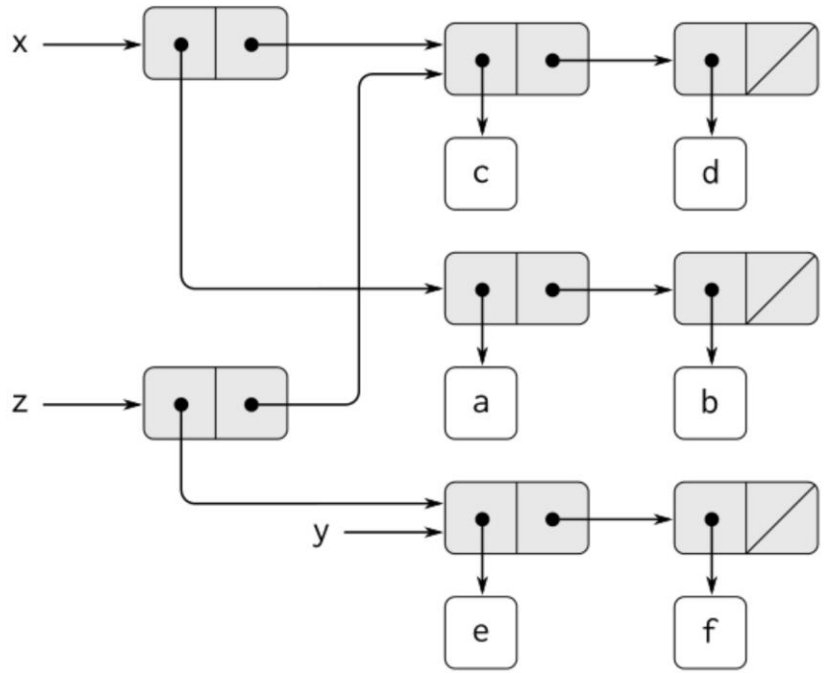


Figure 3.14: Effect of `(define z (cons y (cdr x)))` on the lists in Figure 3.12.

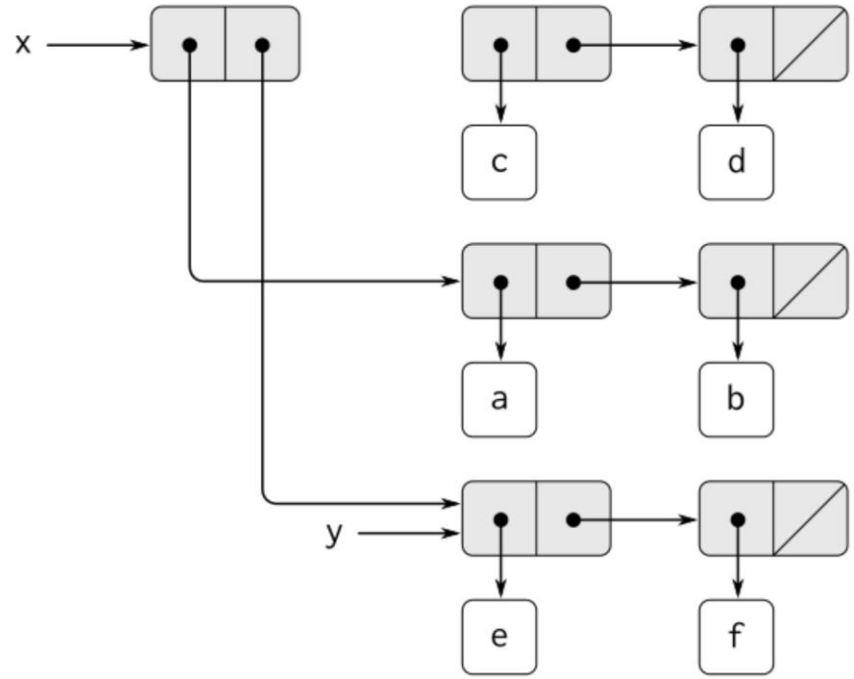


Figure 3.15: Effect of `(set-cdr! x y)` on the lists in Figure 3.12.

Exercise 3.12: The following procedure for appending lists was introduced in Section 2.2.1:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

append forms a new list by successively consing the elements of x onto y. The procedure append! is similar to append, but it is a mutator rather than a constructor. It appends the lists by splicing them together, modifying the final pair of x so that its cdr is now y. (It is an error to call append! with an empty x.)

```
(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)
```

Here `last-pair` is a procedure that returns the last pair in its argument:

```
(define (last-pair x)
  (if (null? (cdr x)) x (last-pair (cdr x))))
```

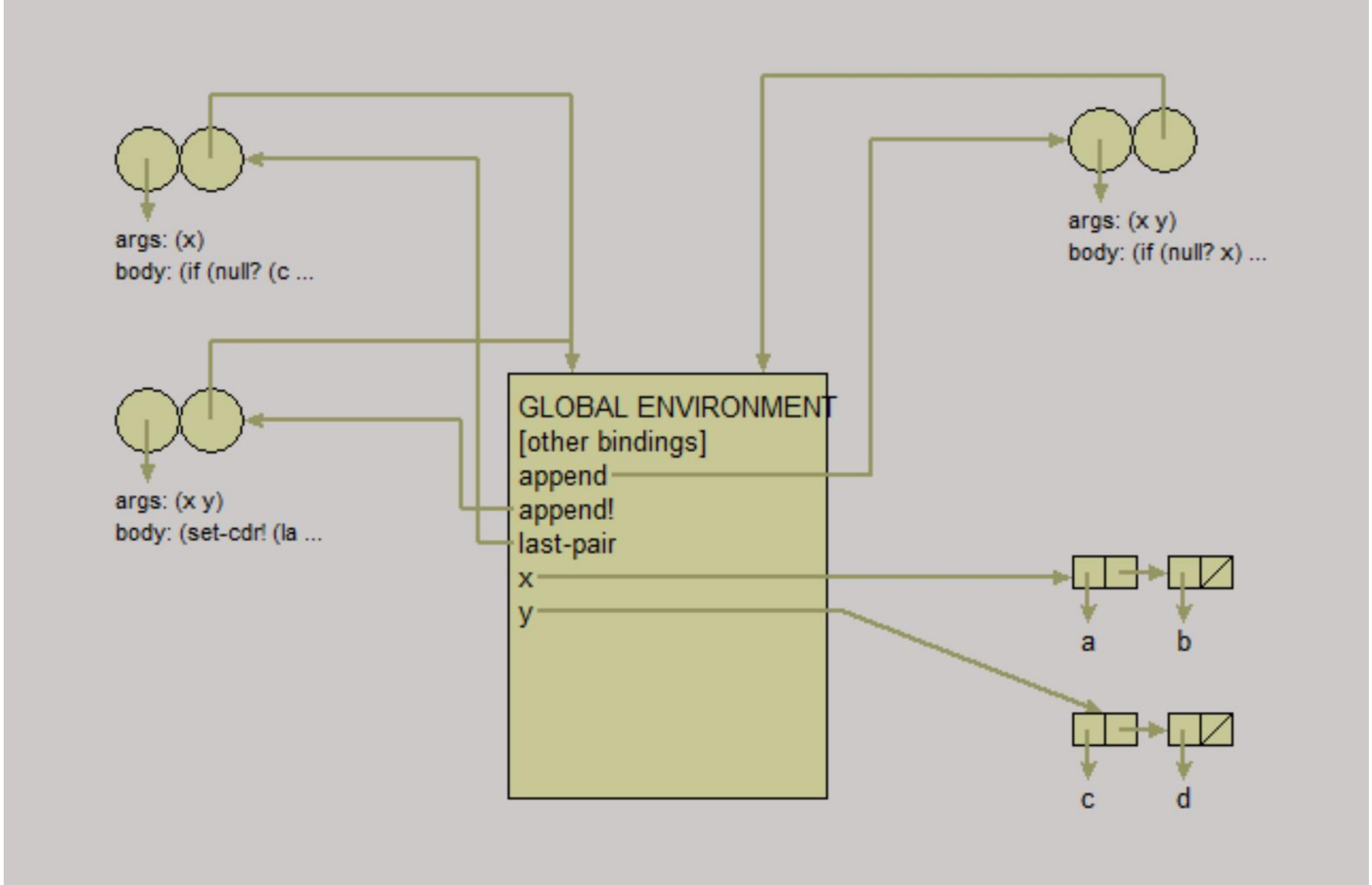
Consider the interaction

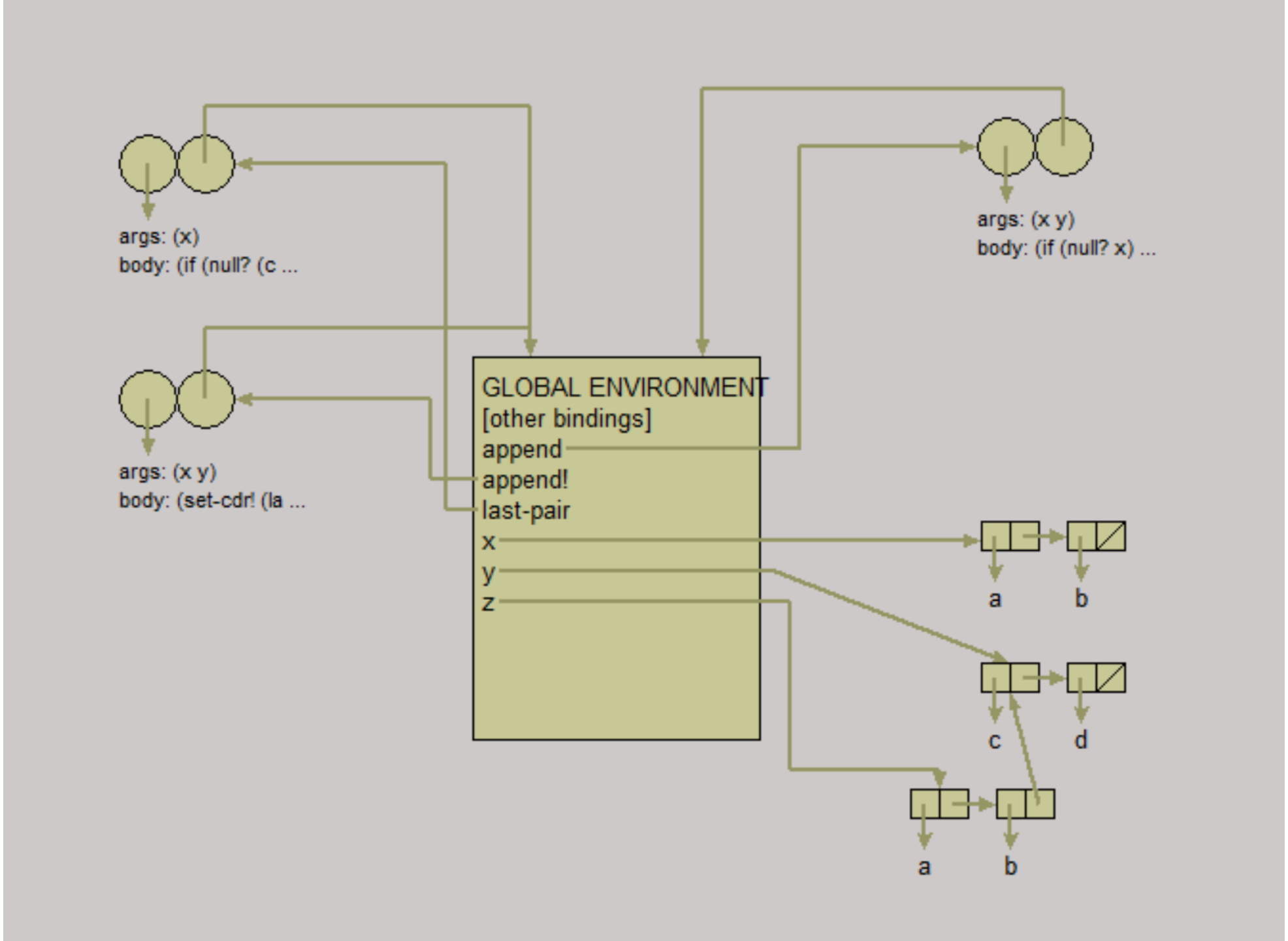
```
(define x (list 'a 'b))
(define y (list 'c 'd))
(define z (append x y))
z
(a b c d)
(cdr x)
⟨response⟩
(define w (append! x y))
w
(a b c d)
(cdr x)
⟨response⟩
```

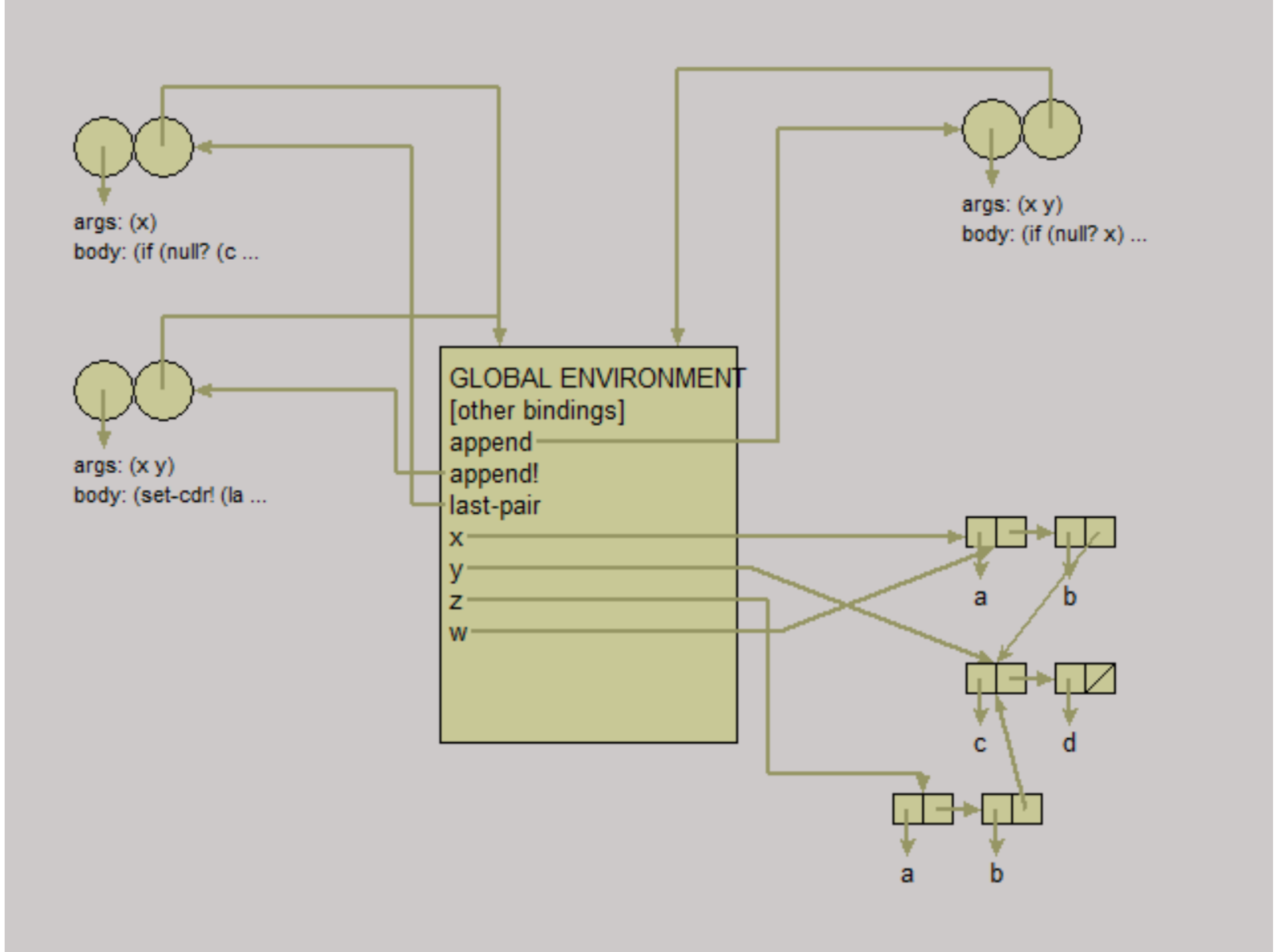
'(b)

'(b c d)

What are the missing ⟨*response*⟩s? Draw box-and-pointer diagrams to explain your answer.


args: (x)
body: (if (null? (c ...
args: (x y)
body: (if (null? x) ...
args: (x y)
body: (set-cdr! (la ...
GLOBAL ENVIRONMENT
[other bindings]
append
append!
last-pair
x
y
a
b
c
d


args: (x)
body: (if (null? (c ...
args: (x y)
body: (if (null? x) ...
args: (x y)
body: (set-cdr! (la ...
GLOBAL ENVIRONMENT
[other bindings]
append
append!
last-pair
x
y
z
a
b
c
d
a
b


args: (x)
body: (if (null? (c ...
args: (x y)
body: (if (null? x) ...
args: (x y)
body: (set-cdr! (la ...
GLOBAL ENVIRONMENT
[other bindings]
append
append!
last-pair
x
y
z
w
a
b
c
d
a
b

```
(mystery '(1 2 3))

x: 1 2 3  y: ()
temp: 2 3
set-cdr! -> x: 1

x: 2 3  y: 1
temp: 3
set-cdr! -> x: 2 1

x: 3  y: 2 1
temp: ()
set-cdr! -> x: 3 2 1

x: ()  y: 3 2 1
```

Exercise 3.14: The following procedure is quite useful, although obscure:

```
(define (mystery x)
  (define (loop x y)
    (if (null? x)
        y
        (let ((temp (cdr x)))
          (set-cdr! x y)
          (loop temp x))))
  (loop x '()))
```

reverses list

`loop` uses the "temporary" variable `temp` to hold the old value of the `cdr` of `x`, since the `set-cdr!` on the next line destroys the `cdr`. Explain what `mystery` does in general. Suppose `v` is defined by `(define v (list 'a 'b 'c 'd))`. Draw the box-and-pointer diagram that represents the list to which `v` is bound. Suppose that we now evaluate `(define w (mystery v))`. Draw box-and-pointer diagrams that show the structures `v` and `w` after evaluating this expression. What would be printed as the values of `v` and `w`?

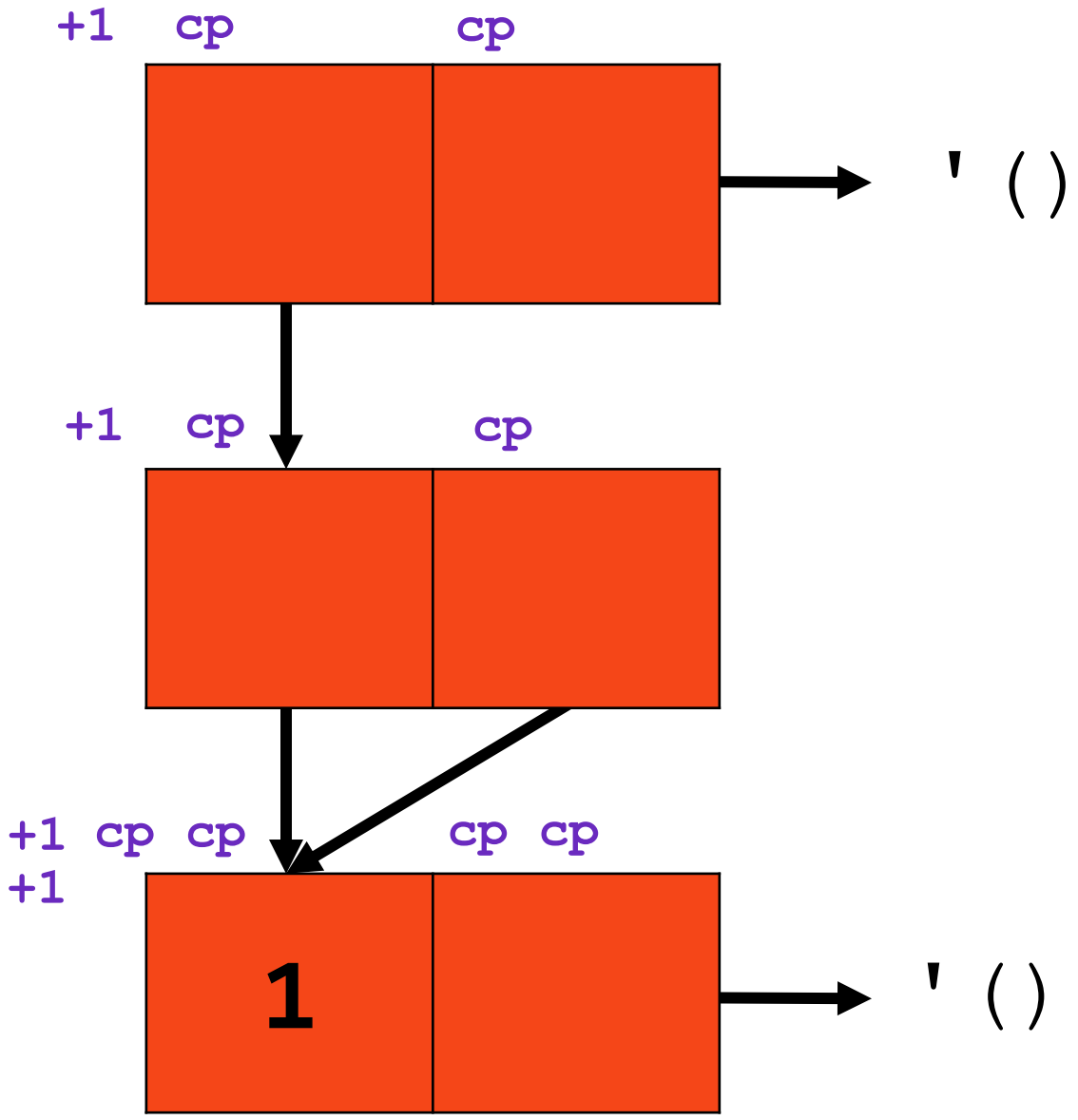
Exercise 3.16: Ben Bitdiddle decides to write a procedure to count the number of pairs in any list structure. "It's easy," he reasons. "The number of pairs in any structure is the number in the `car` plus the number in the `cdr` plus one more to count the current pair." So Ben writes the following procedure:

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

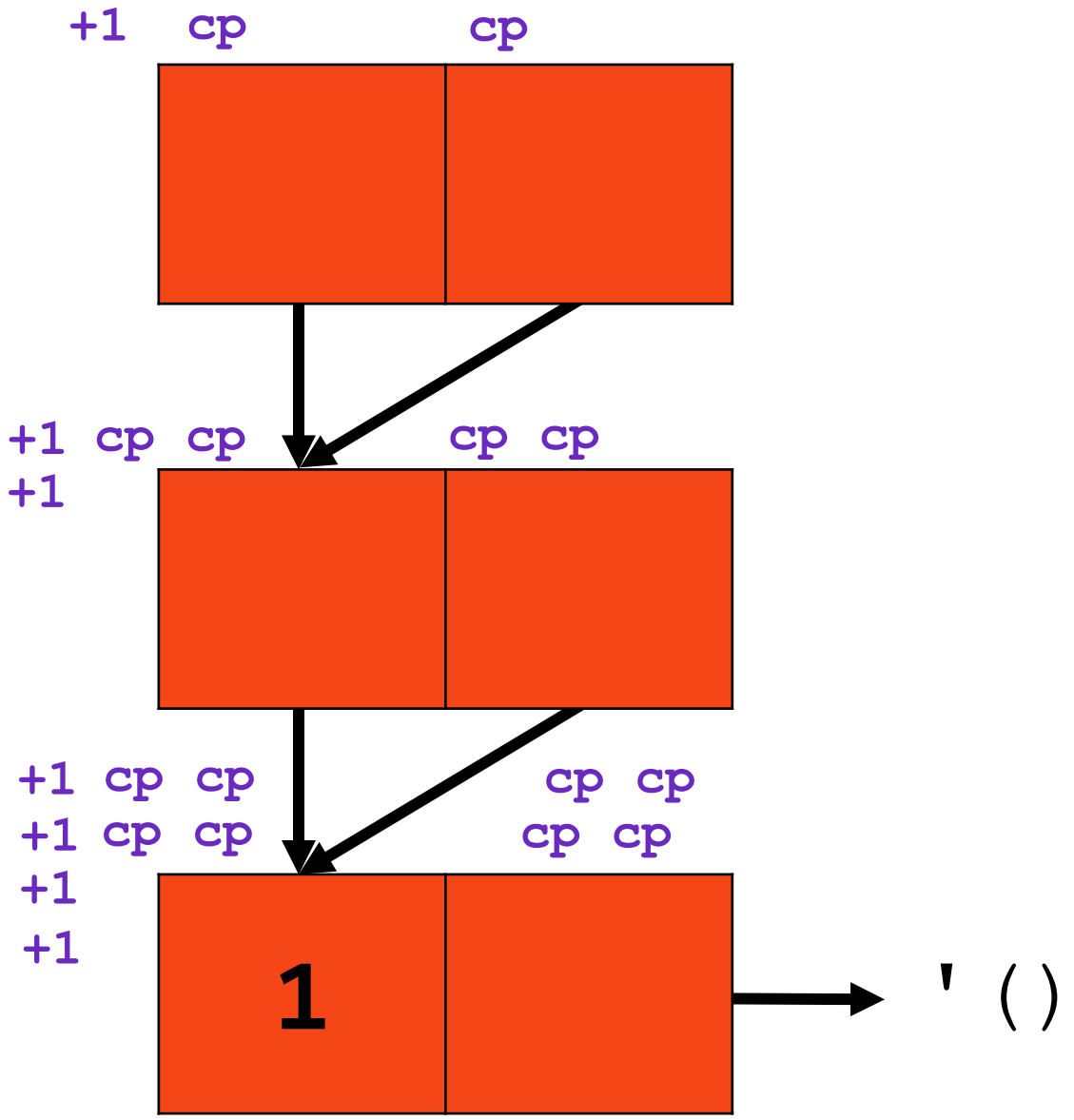
Show that this procedure is not correct. In particular, draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Ben's procedure would return 3; return 4; return 7; never return at all.

```
;; Exercise 3.16 (350-1)

(count-pairs '(1 2 3)) ; -> 3

(define x (cons 1 '()))
(define y (cons x x))
(define z (cons y '()))
(count-pairs z) ; -> 4

(define x (cons 1 '()))
(define y (cons x x))
(define z (cons y y))
(count-pairs z) ; -> 7
```

```
(define x (cons 1 '()))
(define y (cons x x))
(define z (cons y '()))
(count-pairs z) ; -> 4
```



```
(define x (cons 1 '()))
(define y (cons x x))
(define z (cons y y))
(count-pairs z) ; -> 7
```



Exercise 3.17: Devise a correct version of the `count-pairs` procedure of Exercise 3.16 that returns the number of distinct pairs in any structure. (Hint: Traverse the structure, maintaining an auxiliary data structure that is used to keep track of which pairs have already been counted.)

```
;; Exercise 3.17 (page 351)

(define seen '())

(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         (if (memq x seen)
             0
             (begin (set! seen (cons x seen)) 1)))))
```

```
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Undefined operation: CONS" m))))
  dispatch)
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
```

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else
           (error "Undefined operation: CONS" m))))
  dispatch)
```

Operation	Resulting Queue
`(define q (make-queue))`	
`(insert-queue! q 'a)`	`a`
`(insert-queue! q 'b)`	`a b`
`(delete-queue! q)`	`b`
`(insert-queue! q 'c)`	`b c`
`(insert-queue! q 'd)`	`b c d`
`(delete-queue! q)`	`c d`

Figure 3.18: Queue operations.

Because a queue is a sequence of items, we could certainly represent it as an ordinary list; the front of the queue would be the `car` of the list, inserting an item in the queue would amount to appending a new element at the end of the list, and deleting an item from the queue would just be taking the `cdr` of the list. However, this representation is inefficient, because in order to insert an item we must scan the list until we reach the end. Since the only method we have for scanning a list is by successive `cdr` operations, this scanning requires $\Theta(n)$ steps for a list of n items. A simple modification to the list representation overcomes this disadvantage by allowing the queue operations to be implemented so that they require $\Theta(1)$ steps; that is, so that the number of steps needed is independent of the length of the queue.

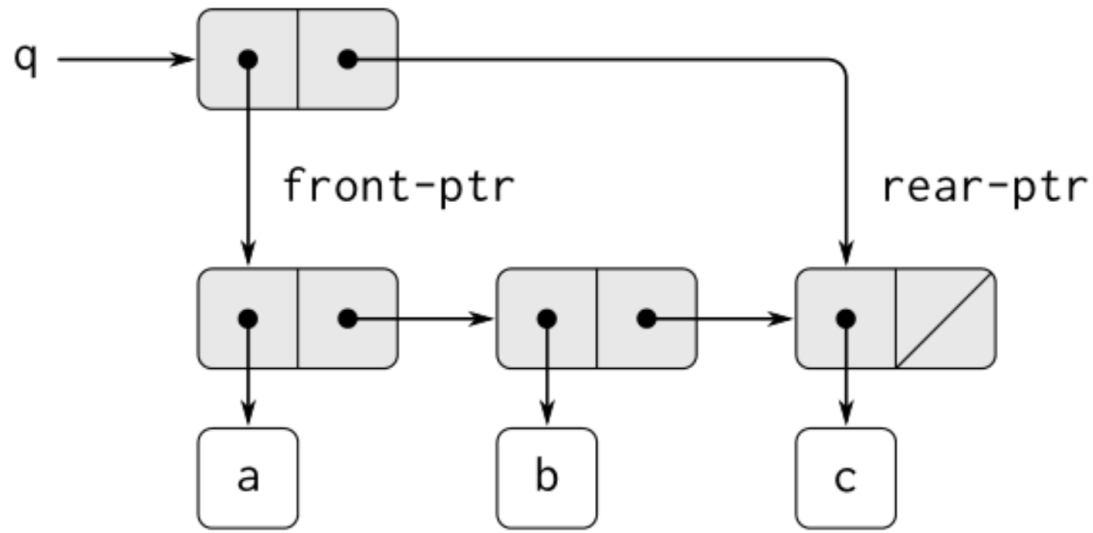


Figure 3.19: Implementation of a queue as a list with front and rear pointers.

```scheme
(define (front-ptr queue) (car queue))
(define (rear-ptr  queue) (cdr queue))
(define (set-front-ptr! queue item)
  (set-car! queue item))
(define (set-rear-ptr!  queue item)
  (set-cdr! queue item))
```

```scheme
(define (empty-queue? queue)
  (null? (front-ptr queue)))
```

```scheme
(define (make-queue) (cons '() '()))
```

```
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))
```

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))
```

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else (set-front-ptr! queue (cdr (front-ptr queue)))
              queue)))
```

Exercise 3.21: Ben Bitdiddle decides to test the queue implementation described above. He types in the procedures to the Lisp interpreter and proceeds to try them out:

```
(define q1 (make-queue))
(insert-queue! q1 'a)
((a) a)
(insert-queue! q1 'b)
((a b) b)
(delete-queue! q1)
((b) b)
(delete-queue! q1)
(() b)
```

"It's all wrong!" he complains. "The interpreter's response shows that the last item is inserted into the queue twice. And when I delete both items, the second `b` is still there, so the queue isn't empty, even though it's supposed to be." Eva Lu Ator suggests that Ben has misunderstood what is happening. "It's not that the items are going into the queue twice," she explains. "It's just that the standard Lisp printer doesn't know how to make sense of the queue representation. If you want to see the queue printed correctly, you'll have to define your own print procedure for queues." Explain what Eva Lu is talking about. In particular, show why Ben's examples produce the printed results that they do. Define a procedure `print-queue` that takes a queue as input and prints the sequence of items in the queue.

```
;; Exercise 3.21 (page 359)

(define q1 (make-queue))
(insert-queue! q1 'a) ;((a) a)
(insert-queue! q1 'b) ;((a b) b)
(insert-queue! q1 'c) ;((a b c) c)
(delete-queue! q1)    ;((b c) c)
(delete-queue! q1)    ;((c) c)

;; front-ptr essentially grows as a list to represent the queue, back-ptr is
;; just for inserting. print should just print front-ptr
```

```scheme
(define (print-queue queue) (front-ptr queue))

;; then just update insert and delete

(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           (print-queue queue))
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           (print-queue queue)))))

(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else (set-front-ptr! queue (cdr (front-ptr queue)))
              (print-queue queue))))
```

Exercise 3.23: A *deque* ("double-ended queue") is a sequence in which items can be inserted and deleted at either the front or the rear. Operations on deques are the constructor `make-deque`, the predicate `empty-deque?`, selectors `front-deque` and `rear-deque`, mutators `front-insert-deque!`, `rear-insert-deque!`, `front-delete-deque!`, and `rear-delete-deque!`. Show how to represent deques using pairs, and give implementations of the operations.[23] All operations should be accomplished in $\Theta(1)$ steps.

```
;; Exercise 3.23 (this is an incorrect solution as I didn't use a doubly-linked list
;;                meaning that the big-O of pop-back-deque is O(n)

(define (front-ptr deque) (car deque))
(define (rear-ptr deque)  (cdr deque))
(define (set-front-ptr! deque item) (set-car! deque item))
(define (set-rear-ptr!  deque item) (set-cdr! deque item))

(define (make-deque) (cons '() '()))

(define (empty-deque? deque) (null? (front-ptr deque)))

(define (front-deque deque)
  (if (empty-deque? deque)
      (error "FRONT called with an empty deque" deque)
      (car (front-ptr deque))))
```

```
(define (back-deque deque)
  (if (empty-deque? deque)
      (error "BACK called with an empty deque" deque)
      (car (rear-ptr deque))))

(define (print-deque deque)
  (if (empty-deque? deque)
      "EMPTY"
      (front-ptr deque)))
```

```
(define (push-back-deque! deque item)
  (let ((new-pair (cons item '())))
    (cond ((empty-deque? deque)
           (set-front-ptr! deque new-pair)
           (set-rear-ptr! deque new-pair)
           (print-deque deque))
          (else
           (set-cdr! (rear-ptr deque) new-pair)
           (set-rear-ptr! deque new-pair)
           (print-deque deque)))))
```

```
(define (push-front-deque! deque item)
  (let ((new-pair (cons item '())))
    (cond ((empty-deque? deque)
           (set-front-ptr! deque new-pair)
           (set-rear-ptr! deque new-pair)
           (print-deque deque))
          (else
           (set-cdr! new-pair (front-ptr deque))
           (set-front-ptr! deque new-pair)
           (print-deque deque)))))
```

```
(define (get-second-last-pair lst)
  (if (null? (cdr (cdr lst)))
      lst
      (get-second-last-pair (cdr lst))))

(define (pop-back-deque! deque)
  (cond ((empty-deque? deque)
         (error "POP-BACK! called with an empty deque" deque))
        ((= (length (front-ptr deque)) 1) (set! deque (cons '() '()))
                                          (print-deque deque))
        (else (set-rear-ptr! deque (get-second-last-pair (front-ptr deque)))
              (set-cdr! (get-second-last-pair (front-ptr deque)) '())
              (print-deque deque))))
```

```
(define q (make-deque))
(push-back-deque! q 'a)  ; (a)
(push-back-deque! q 'b)  ; (a b)
(push-back-deque! q 'c)  ; (a b c)
(front-deque q)          ; 'a
(back-deque q)           ; 'c
(pop-front-deque! q)     ; (b c)
(pop-front-deque! q)     ; (c)
(push-front-deque! q 'a) ; (a c)
(push-front-deque! q 'b) ; (b a c)
(pop-back-deque! q)      ; (b a)
(pop-back-deque! q)      ; (b)
(pop-back-deque! q)      ; ()
(push-back-deque! q 'a)  ; (b a) <- fails
```

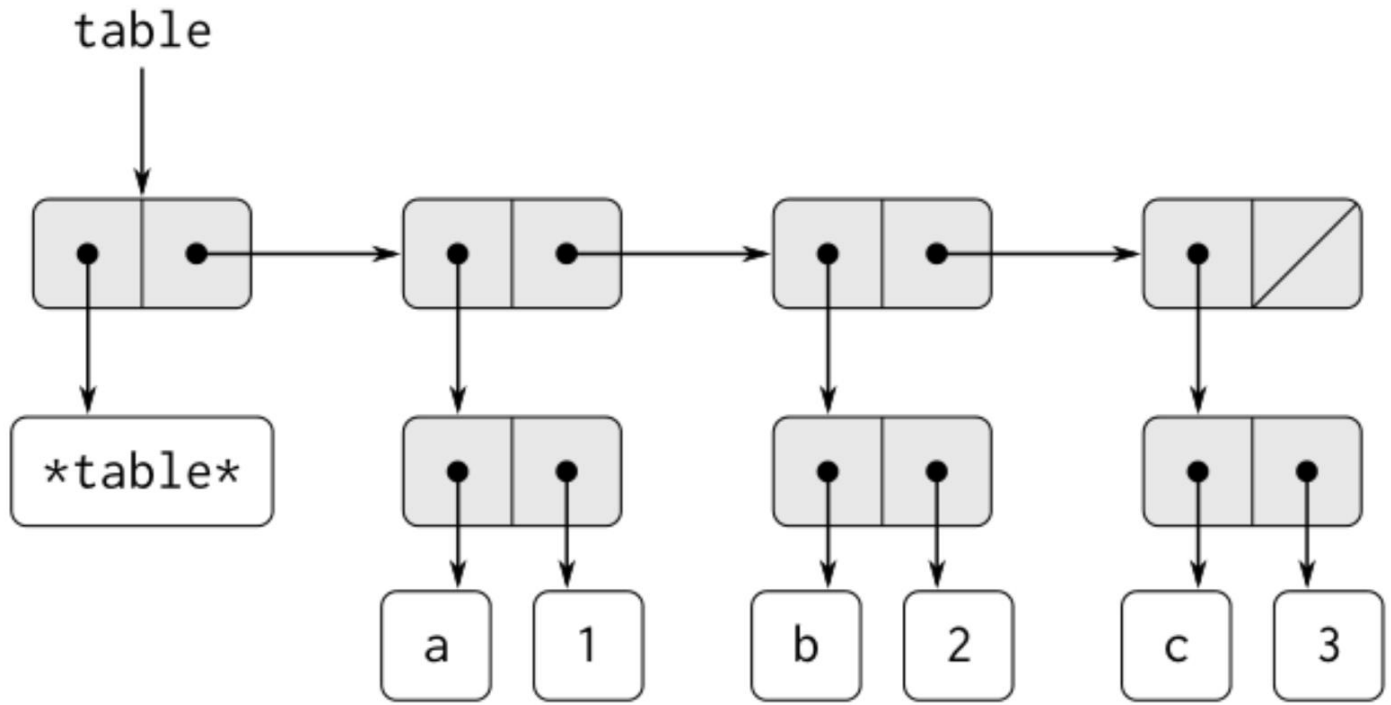


Figure 3.22: A table represented as a headed list.

```scheme
;; 1D Table from book

(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        false)))

(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))

(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value)
                        (cdr table)))))
  'ok)

(define (make-table)
  (list '*table*))
```

```
(define t (make-table))
(insert! 'a 1 t) ; ok
(insert! 'b 2 t) ; ok
(insert! 'c 3 t) ; ok
(insert! 'd 4 t) ; ok
t ; (*table* (d . 4) (c . 3) (b . 2) (a . 1))
```

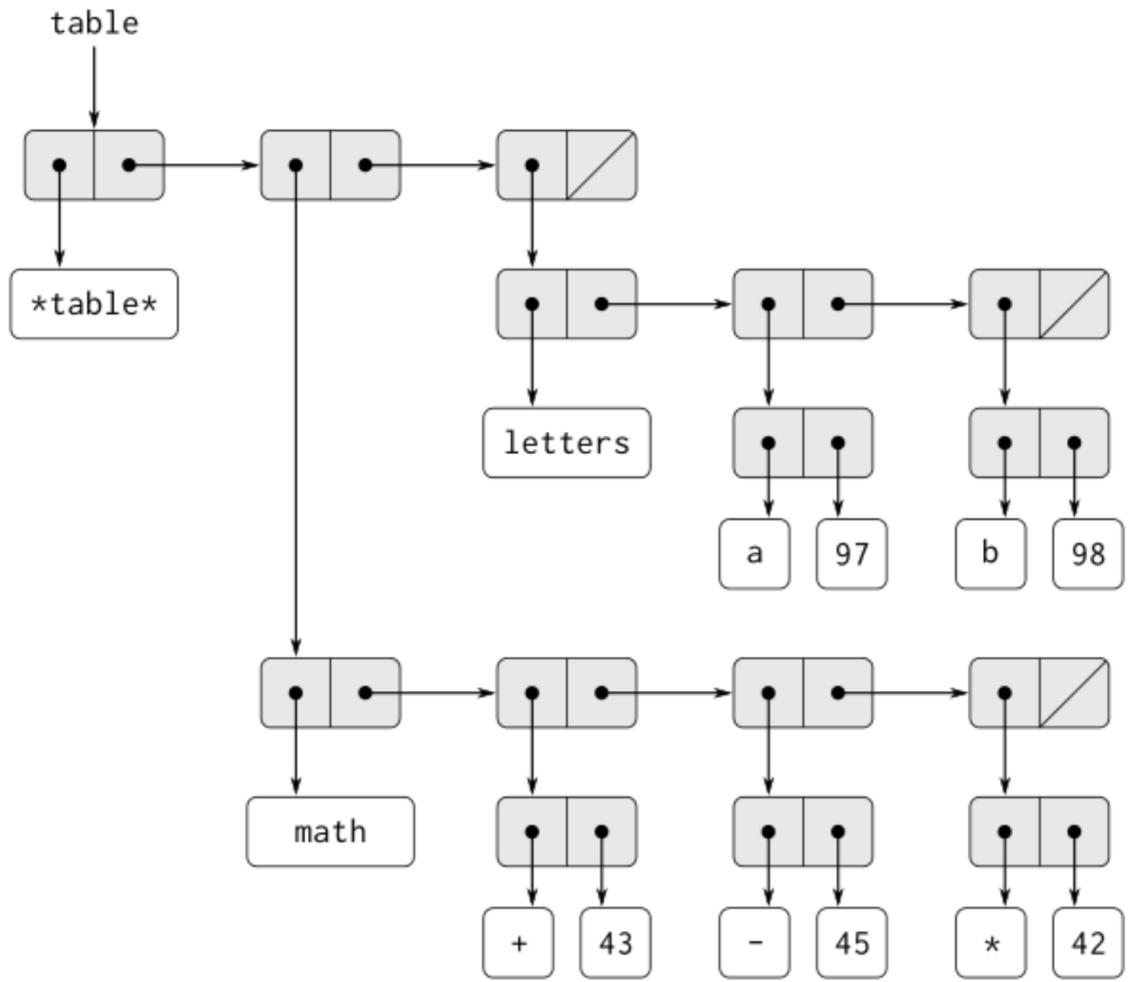


Figure 3.23: A two-dimensional table.

```scheme
;; 2D Table from book (non-message passing style)

(define (lookup key-1 key-2 table)
  (let ((subtable
         (assoc key-1 (cdr table))))
    (if subtable
        (let ((record
               (assoc key-2 (cdr subtable))))
          (if record
              (cdr record)
              false))
        false)))

(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable)))))
        (set-cdr! table
                  (cons (list key-1
                              (cons key-2 value))
                        (cdr table)))))
  'ok)
```

```
(define t (make-table))
(insert! 'a 'a 1 t) ; ok
(insert! 'a 'b 2 t) ; ok
(lookup  'a '1 t)   ; #f
(lookup  'a 'a t)   ; 1
```

```scheme
;; 2D Table from book (message passing style)

(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2)
      (let ((subtable
             (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record
                   (assoc key-2 (cdr subtable))))
              (if record (cdr record) false))
            false)))
    (define (insert! key-1 key-2 value)
      (let ((subtable
             (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record
                   (assoc key-2 (cdr subtable))))
              (if record
                  (set-cdr! record value)
                  (set-cdr! subtable
                            (cons (cons key-2 value)
                                  (cdr subtable)))))
            (set-cdr! local-table
                      (cons (list key-1 (cons key-2 value))
                            (cdr local-table)))))
      'ok)
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unknown operation: TABLE" m))))
    dispatch))
```

```
(define t (make-table))
((t 'insert-proc!) 'a 'a 1) ; ok
((t 'insert-proc!) 'a 'b 2) ; ok
((t 'lookup-proc) 'a '1)    ; #f
((t 'lookup-proc) 'a 'a)    ; 1
```

Exercise 3.25: Generalizing one- and two-dimensional tables, show how to implement a table in which values are stored under an arbitrary number of keys and different values may be stored under different numbers of keys. The `lookup` and `insert!` procedures should take as input a list of keys used to access the table.

```scheme
;; Exercise 3.25 (page 367)

(define (lookup keys table)
  (let ((record (assoc keys (cdr table))))
    (if record
        (cdr record)
        false)))

(define (insert! keys value table)
  (let ((record (assoc keys (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons keys value)
                        (cdr table)))))
  'ok)

(define (make-table)
  (list '*table*))

(define t (make-table))
(insert! '(a b c) 1 t) ; ok
(insert! '(a b) 2 t)   ; ok
(lookup  '(a 1) t)     ; #f
(lookup  '(a b) t)     ; 2
```

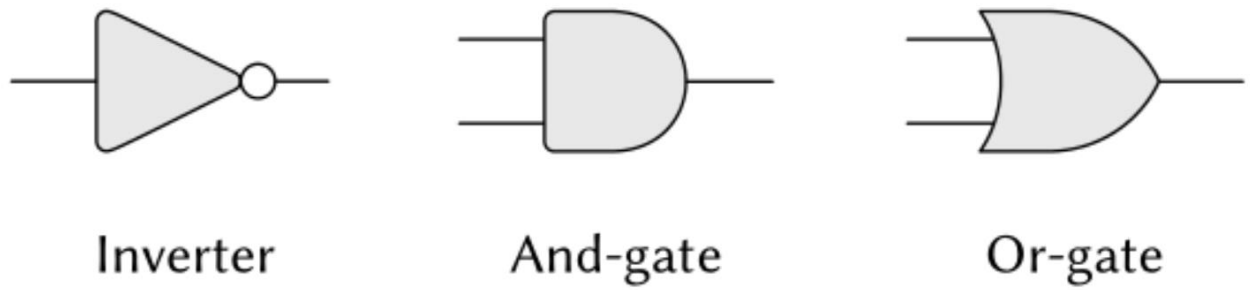


Figure 3.24: Primitive functions in the digital logic simulator.

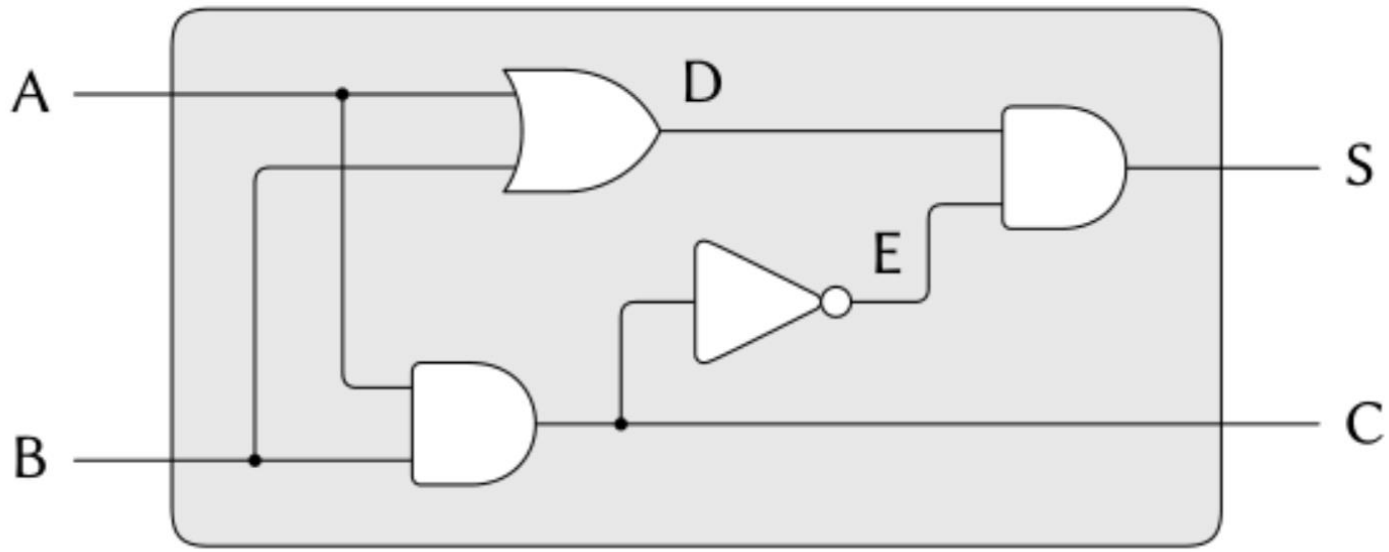


Figure 3.25: A half-adder circuit.

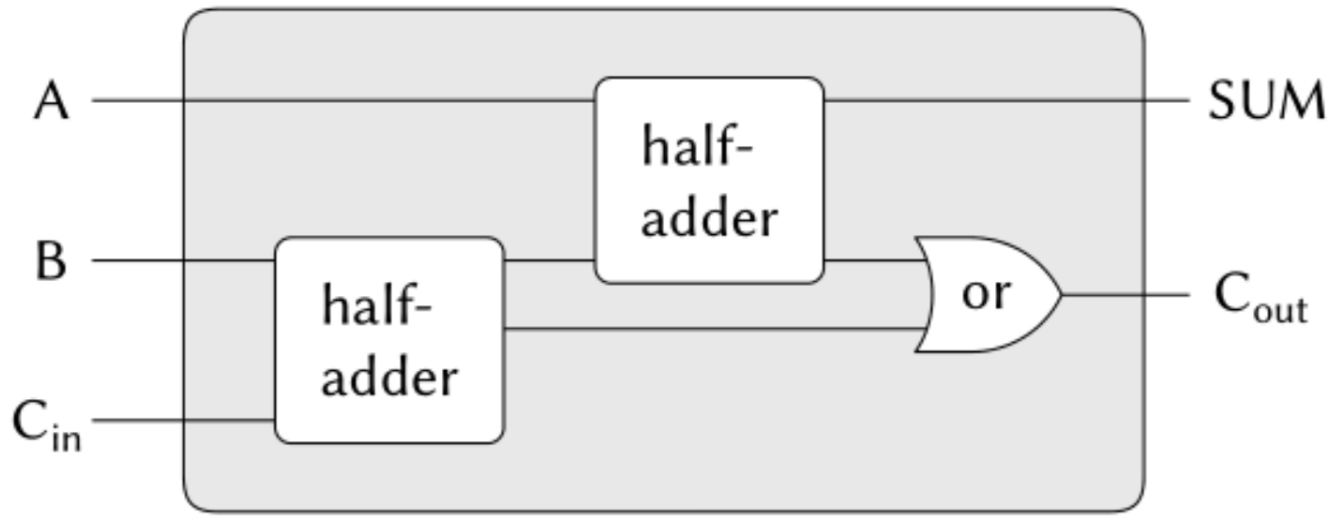


Figure 3.26: A full-adder circuit.

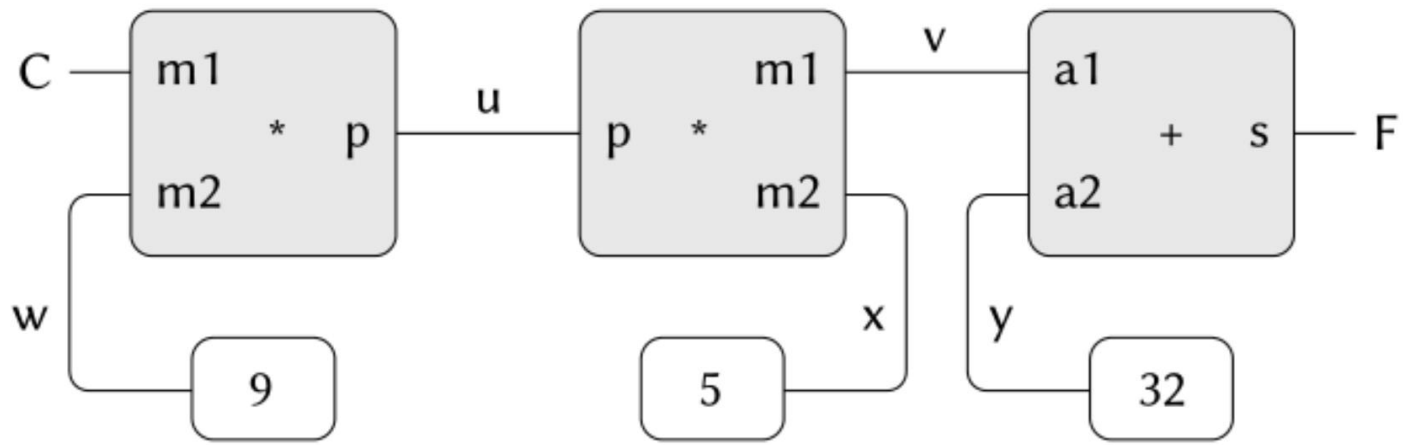


Figure 3.28: The relation $9C = 5(F - 32)$ expressed as a constraint network.

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

Primitives and Means of Combination

```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
(define d (make-wire))
(define e (make-wire))
(define s (make-wire))
```

Inverter

And-gate

```
(or-gate a b d)
(and-gate a b c)
(inverter c e)
(and-gate d e s)
```

QUEUE

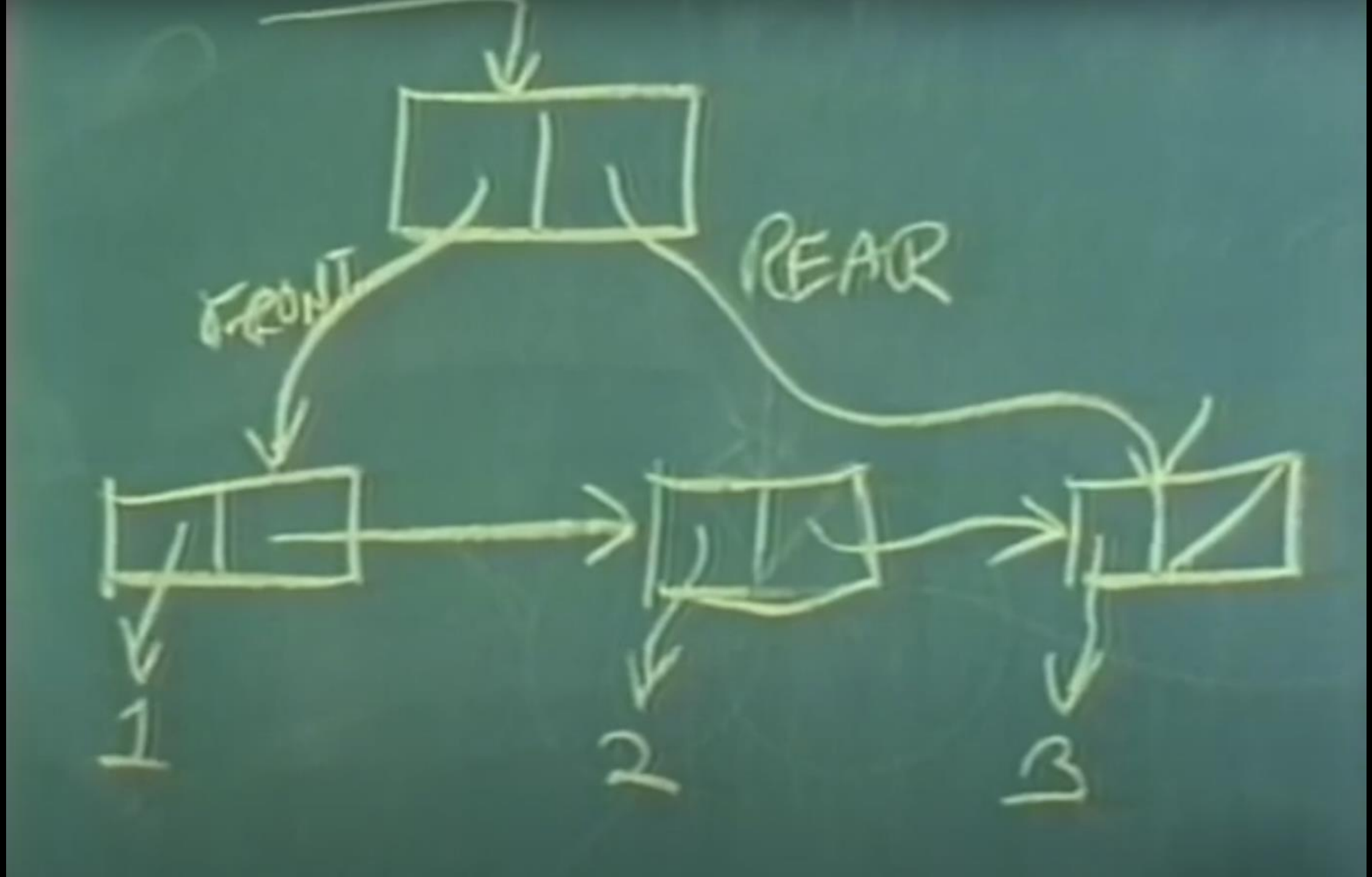
(MAKE-QUEUE) ⟹ new queue

(INSERT-QUEUE! queue item)

(DELETE-QUEUE! queue)

(FRONT-QUEUE queue)

(EMPTY-QUEUE? queue


FRONT
REAR
1
2
3

```
(DEFINE (CONS X Y)
  (λ (M) (M X Y)))

(DEFINE (CAR X)
  (X (λ (A D) A)))

(DEFINE (CDR X)
  (X (λ (A D) D)))
```

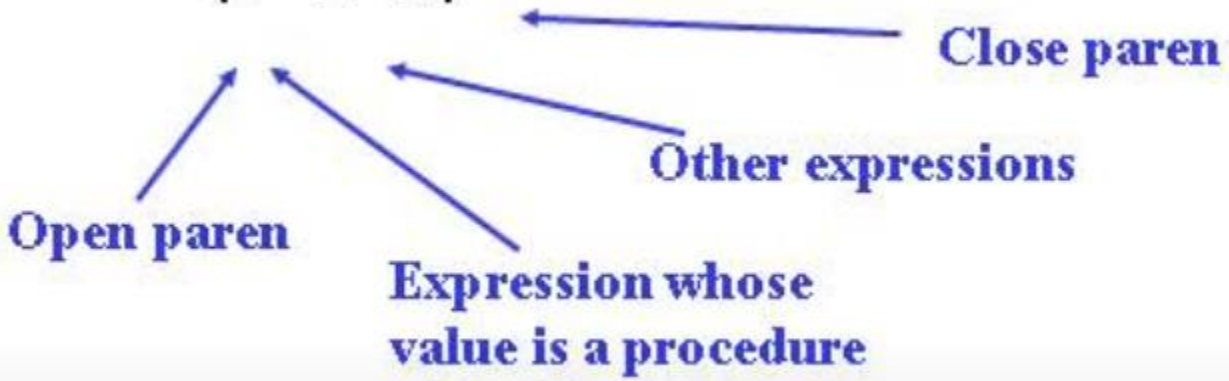
"Lambda Calculus" Mutable Data

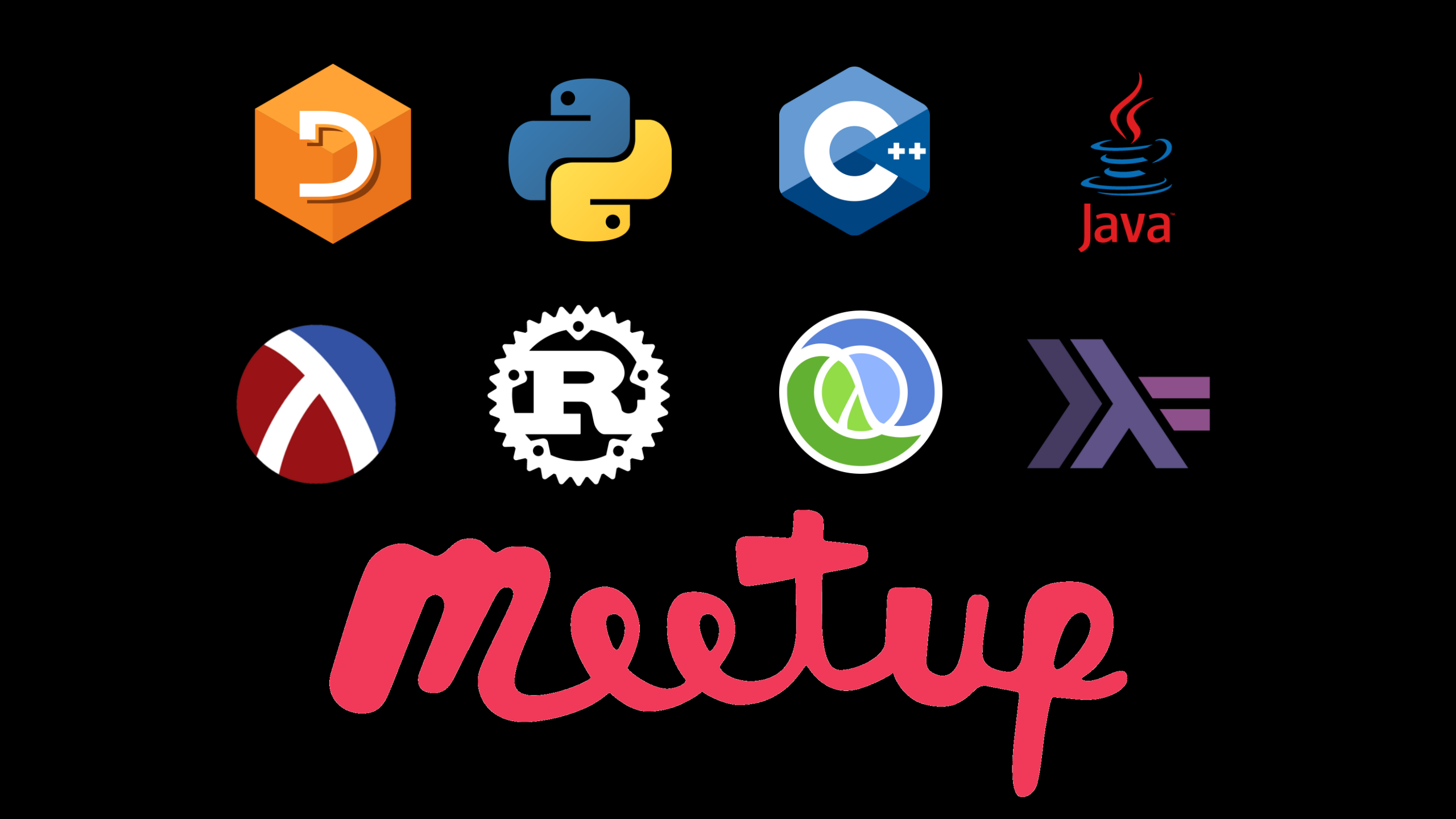
```
(define (cons x y)
  (lambda (m)
    (m x
       y
       (lambda (n) (set! x n))
       (lambda (n) (set! y n)))))
```

Language elements – combinations

- How do we create expressions using these procedures?

```
(+ 2 3)
```




Java
meetup