```
; Let's write quicksort... First, a data defn.

; A num-list is either
; - empty, or
; - (cons num num-list)

; Based on our knowledge of the algorithm, we know we need helper
; functions. Let's write them first.

(define (nl-proc pivot nl)
  (cond
    [(empty? nl) ...]
    [(cons? nl)
     ... (first nl) ...
     ... (nl-proc pivot (rest nl)) ...]))

; We need helper functions to split the list
;  for a pivot item

; nums-le : num num-list -> num-list

(define (nums-le pivot nl)
  (cond
    [(empty? nl) ...]
    [(cons? nl)
     (cond
       [(<= pivot (first nl))
        ... (first nl) ...
        ... (nums-le pivot (rest nl)) ...]
       [else
        ... (first nl) ...
        ... (nums-le pivot (rest nl)) ...])]))

(define (nums-le pivot nl)
  (cond
    [(empty? nl) empty]
    [(cons? nl)
     (cond
       [(<= (first nl) pivot)
        (cons (first nl) (nums-le pivot (rest nl)))]
       [else
        (nums-le pivot (rest nl))])]))

;; The cut-and-paste approach above seems wrong,
;; especially since the functions are so similar.
;; What if we wrote one function and made the operator
;; an argument?
```

```
(define (nums-filter pivot nl COMPARATOR)
  (cond
    [(empty? nl) empty]
    [(cons? nl)
     (cond
       [(COMPARATOR (first nl) pivot)
        (cons (first nl) (nums-filter pivot (rest nl) COMPARATOR))]
       [else
        (nums-filter pivot (rest nl) COMPARATOR)])]))

; This works! Here's the contract for num-filter:
;   nums-filter : num num-list (num num -> boolean) -> num-list


(define (nums-le pivot nl)
  (nums-filter pivot nl <=))

(define (nums-gt pivot nl)
  (nums-filter pivot nl >))

>(nums-le 5 empty)
empty
>(nums-le 5 (cons 1 empty))
(cons 1 empty)
>(nums-le 5 (cons 5 empty))
(cons 5 empty)
>(nums-le 5 (cons 6 empty))
empty
>(nums-le 5 (cons 1 (cons 6 (cons 5 empty))))
(cons 1 (cons 5 empty))

; Now the main function.
;  quicksort : num-list -> num-list
;  returns a list containing the same numbers, sorted in ascending order

; Template
;(define (quicksort nl)
;  (cond
;    [(empty? nl) ...]
;    [(cons? nl) ... (first nl) ...
;                ... (quicksort (rest nl)) ...]))

; This is a dead end:
; (quicksort (rest nl)) must return a sorted list,
; so the only choice is to insert (first nl) --
; which leads to some variant of insertion sort
```

```
;; Another idea
(define (quicksort nl)
  (cond
    [(empty? nl) ···]
    [else
     (combine-parts
      ··· (quicksort (nums-le nl)) ···
      ···
      ··· (quicksort (nums-gt nl)) ···)]))

;; >> BUGGY << version:
(define (quick-sort nl)
  (cond
    [(empty? nl) empty]
    [else
     (combine-parts
      (quick-sort (nums-le (first nl) nl))
      (first nl)
      (quick-sort (nums-gt (first nl) nl)))]))
;; Doesn't terminate!

;; correct version:
(define (quick-sort nl)
  (cond
    [(empty? nl) empty]
    [else
     (combine-parts
      (quick-sort (nums-le (first nl) (rest nl)))
      (first nl)
      (quick-sort (nums-gt (first nl) (rest nl))))]))

(define (combine-parts le-nums pivot ge-nums)
  (append le-nums (cons pivot ge-nums)))

> (quick-sort empty)
empty
> (quick-sort (cons 1 empty))
(cons 1 empty)
>(quick-sort (cons 1 (cons 2 empty)))
(cons 1 (cons 2 empty))
>(quick-sort (cons 2 (cons 1 empty)))
(cons 1 (cons 2 empty))
>(quick-sort (cons 2 (cons 1 (cons 3 empty))))
(cons 1 (cons 2 (cons 3 empty)))
>(quick-sort (cons 11 (cons 8 (cons 14 (cons 7 empty)))))
(cons 7 (cons 8 (cons 11 (cons 14 empty))))
```

```
;; Recipe for generative recursion:
;; The template is of the form
(define (func problem)
  (cond
   [(trivially-solveable? problem)  ...]
   [else
    (combine-solutions
     (func (generate-problem ...))
     ...
     (func (generate-problem ...)))]))

;; Another exaple of generative recursion: binary sort

;; find-root num num (num -> num) -> num
;; Purpose: finds number n between a and b such that
;;  (f n) is within TOLERANCE of zero. f is an operator,
;;  (f a) must be negative, and (f b) must be positive.
```

```
(define TOLERANCE 0.00005)
(define (find-root a b f)
  (cond
    [(<= (abs (- b a)) TOLERANCE) ···]
    [else
     (combine-parts
      (find-root a (/ (+ a b) 2) f)
      (find-root (/ (+ a b) 2) b f))]))

(define (find-root a b f)
  (cond
    [(<= (abs (- b a)) TOLERANCE) a]
    [else
     (cond
      [(positive? (f (/ (+ a b) 2)))
       (find-root a (/ (+ a b) 2) f)]
      [else
       (find-root (/ (+ a b) 2) b f)])]))

> (find-root −0.5 0.5 sin)
0
> (find-root −2 0 cos)
−1.5712890625
```

```
;; We can make the repeated midpoint calculation,
;;  (/ (+ a b) 2), easier to read by using a local
;;  definition:

(define (find-root a b f)
  (cond
    [(<= (abs (- b a)) TOLERANCE) a]
    [else
     (local [(define mid (/ (+ a b) 2))]
       (cond
         [(positive? (f mid))
          (find-root a mid f)]
         [else
          (find-root mid b f)]))]))

;; What about roots for functions we define?
;; Let's try a polynomial: 4n^2 - 2n - 10

(define (poly n)
  (+ (* 4 n n)
     (* -2 n)
     -10))

> (poly 1)
-8
> (poly 10)
370
> (find-root 1 10 poly)
1.8507537841796875

;; In other words: FUNCTIONS ARE VALUES
```

```
;; A more general number-filtering program:
;;   nums-filter2 : (num -> boolean) num-list -> num-list

(define (nums-filter2 CHECKER nl)
  (cond
    [(empty? nl) empty]
    [(cons? nl)
     (cond
       [(CHECKER (first nl))
        (cons (first nl) (nums-filter2 CHECKER (rest nl)))]
       [else
        (nums-filter2 CHECKER (rest nl))])]))

(define (nums-le pivot nl)
  (local [(define (le-pivot n)
            (<= n pivot))]
    (nums-filter2 le-pivot nl)))

;; Step through the first few evaluations above (using the
;;  new reduction rule for 'local' expressions)

;; How about filtering animals?

;; animal-filter : (animal -> bool) animal-list -> animal-list

(define (animal-filter CHECKER al)
  (cond
    [(empty? al) empty]
    [(cons? al)
     (cond
       [(CHECKER (first al))
        (cons (first al) (animal-filter CHECKER (rest al)))]
       [else
        (animal-filter CHECKER (rest al))])]))

(define (boa-filter al)
  (animal-filter boa? al))

;; See any difference in the above filter programs, beside the names?
```

```
;; Let's just call it "filter"

;; filter : (T -> bool) T-list -> T-list

(define (filter f? l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     (cond
       [(f? (first l))
        (cons (first l) (filter f? (rest l)))]
       [else
        (filter f? (rest l))])]))

;; This function is so useful that it's built into Scheme.

;; Another common thing we do to lists is convert each element
;;   (e.g., convert DM to dollars)

;; map : (T1 -> T2) T1-list -> T2-list

(define (map f l)
  (cond
    [(empty? l) empty]
    [(cons? l) (cons (f (first l))
                     (map f (rest l)))]))

(define (convert rate l)
  (local [(define (convert-one n)
            (* rate n))]
    (map convert-one l)))

> (convert 1.25 (list 4 10))
(list 5 12.5)

;; Functions as values give us even more power: we can
;;  write functions that return functions!
;; This is useful for example, in using Newton's method
;;  to calculate roots. Given a function, we need its
;;  derivative...
```

```
; square :  num -> num

(define (square x)
  (* x x))

; d/dx-square :  num -> num

(define EPSILON 0.01)
(define (d/dx-square n)
  (/ (- (square (+ n EPSILON)) (square (- n EPSILON)))
     (* 2 EPSILON)))

; d/dx-at-n :  (num -> num) num -> num

(define (d/dx-at-n square n)
  (/ (- (square (+ n EPSILON)) (square (- n EPSILON)))
     (* 2 EPSILON)))

; d/dx-at-n :  (num -> num) num -> num

(define (d/dx-at-n f n)
  (/ (- (f (+ n EPSILON)) (f (- n EPSILON)))
     (* 2 EPSILON)))

; d/dx-square :  num -> num

(define (d/dx-square n) (d/dx-at-n square n))

; But we really want d/dx : (num -> num) -> (num -> num)


(define (d/dx f)
  (local [(define (d/dx-f n)
            (/ (- (f (+ n EPSILON)) (f (- n EPSILON)))
               (* 2 EPSILON)))]
     d/dx-f))

; Why can't we return a function wihout using 'local'?
; After all, we can write down numbers, positions, lists, and armadillos directly.
; Functions? We would like to be able to write something like he following.


(define (d/dx f)
  (function (n) ;; <- a function of one argument, n
     (/ (- (f (+ n EPSILON)) (f (- n EPSILON))) ;; <- body
        (* 2 EPSILON))))
```

```
; Almost --- just use 'lambda' instead of 'function':

(define (d/dx f)
  (lambda (n)
     (/ (- (f (+ n EPSILON)) (f (- n EPSILON)))
        (* 2 EPSILON))))

; Why 'lambda'? Tradition.

; We can use anonymous functions when calling d/dx, too:

(define d/dx-square (d/dx (lambda (x) (* x x))))
(define d/dx-cube (d/dx (lambda (x) (* x x x))))

; In fact, the 'define' form you've learned for functions is just a
;  shorthand for 'lambda'-valued abbreviations:

(define square (lambda (x) (* x x)))
;; same as
(define (square x) (* x x))

; convertDM is now a one-liner:

(define (convertDM l) (map (lambda (x) (* x 1.82)) l))

; Of course, abbreviations are just a convenience. It turns out that
; we could simplify the language by getting rid of 'define', and
; just having 'lambda' plus application forms. We don't even need numbers
; to do useful things, because we could just use '(lambda (x) (x))'
; as a long name for '1', etc.

; Our claim has been that computing has nothing to do with computers.
; Now, by making our language more powerful, then throwing out
; redundant syntax, we have arrived at a brand new language ---
; at least, it was brand new in the 1930's! Alonzo Church devised this
; Lambda Calculus long before mechanical computers were widely available.
```