Natural recursion

- The recursion happens where the data is put together
  - the natural place for it
  - recurses on sub-part of data → clear it will always terminate

```
LOX is one of:
- empty
- (cons X LOX)
```

```
(cons 1 (cons 2 (cons 3 … empty
```

```
(define (foo lox)
  (cond [(empty? lox) base]
        [else
          (fn (first lox)
              (fn-for-lox (rest lox)))]))
```

```
(fn 1
    (fn-for-lox (cons 2 (cons 3 … empty
```

subpiece of current data

Generative recursion

- The “non-natural” superset of natural recursion
- Recurses on new data → potentially does not terminate

```
(define (genrec-fn d)
  (cond [(trivial? d) (trivial-answer d)]
        [else
          (... d
               (genrec-fn (next-problem d)))]))
```

Quicksort

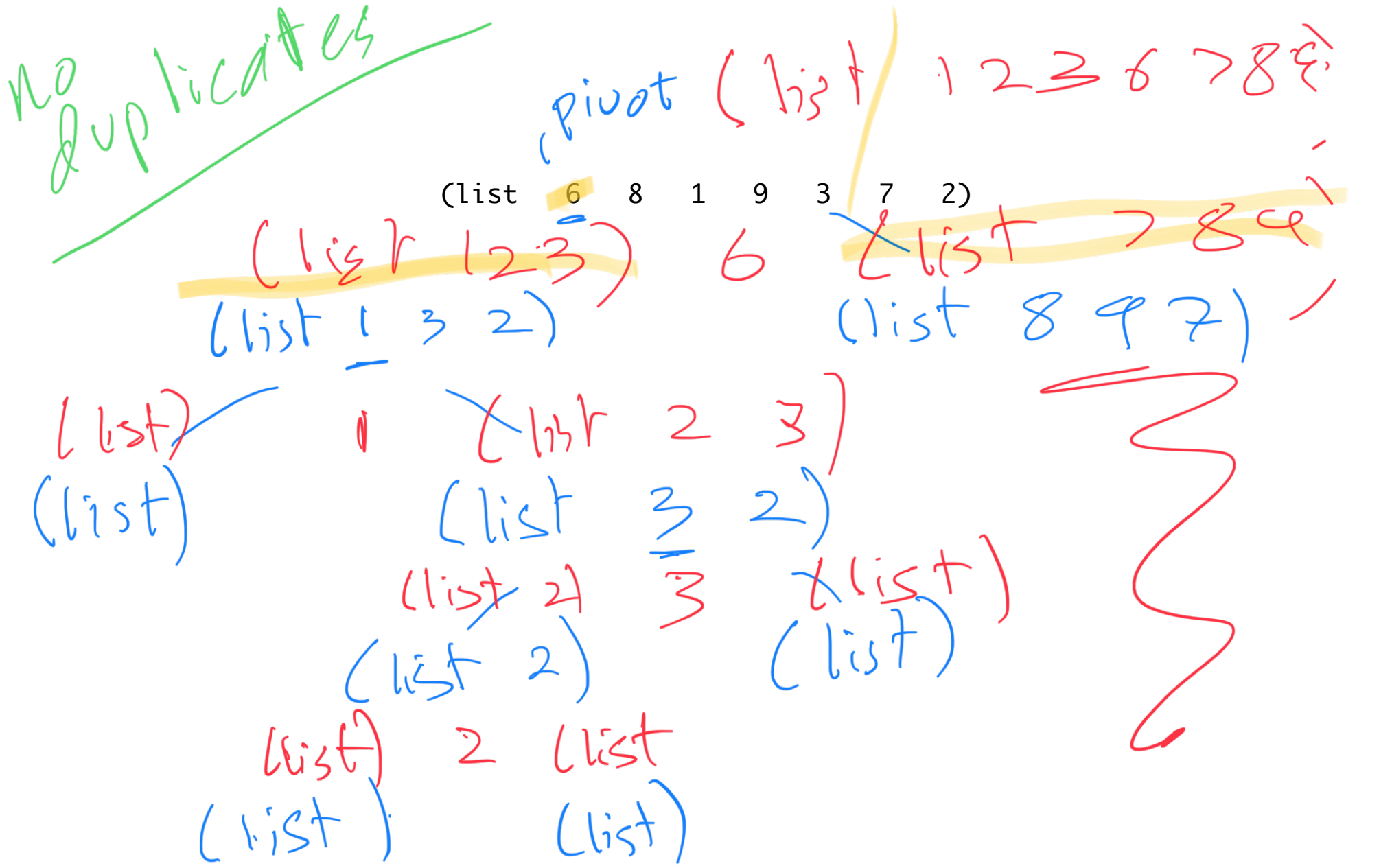
From Wikipedia, the free encyclopedia

Quicksort is an ~~in-place~~ sorting algorithm. Developed by British computer scientist Tony Hoare in 1959[1] and published in 1961,[2] it is still a commonly used algorithm for sorting. When implemented well, it can be somewhat faster than merge sort and about two or three times faster than heapsort.[3][*contradictory*]

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called **partition-exchange sort**.[4] The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Efficient implementations of Quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved.

Mathematical analysis of quicksort shows that, on average, the algorithm takes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons.


no duplicates
(pivot (list 1 2 3 6 7 8 9)
(list 6 8 1 9 3 7 2)
(list 1 2 3) 6 (list 7 8 9)
(list 1 3 2) (list 8 9 7)
(list) 1 (list 2 3)
(list) (list 3 2)
(list 2) 3 (list)
(list 2) (list)
(list) 2 (list
(list) (list)

Back to Design Recipes Table

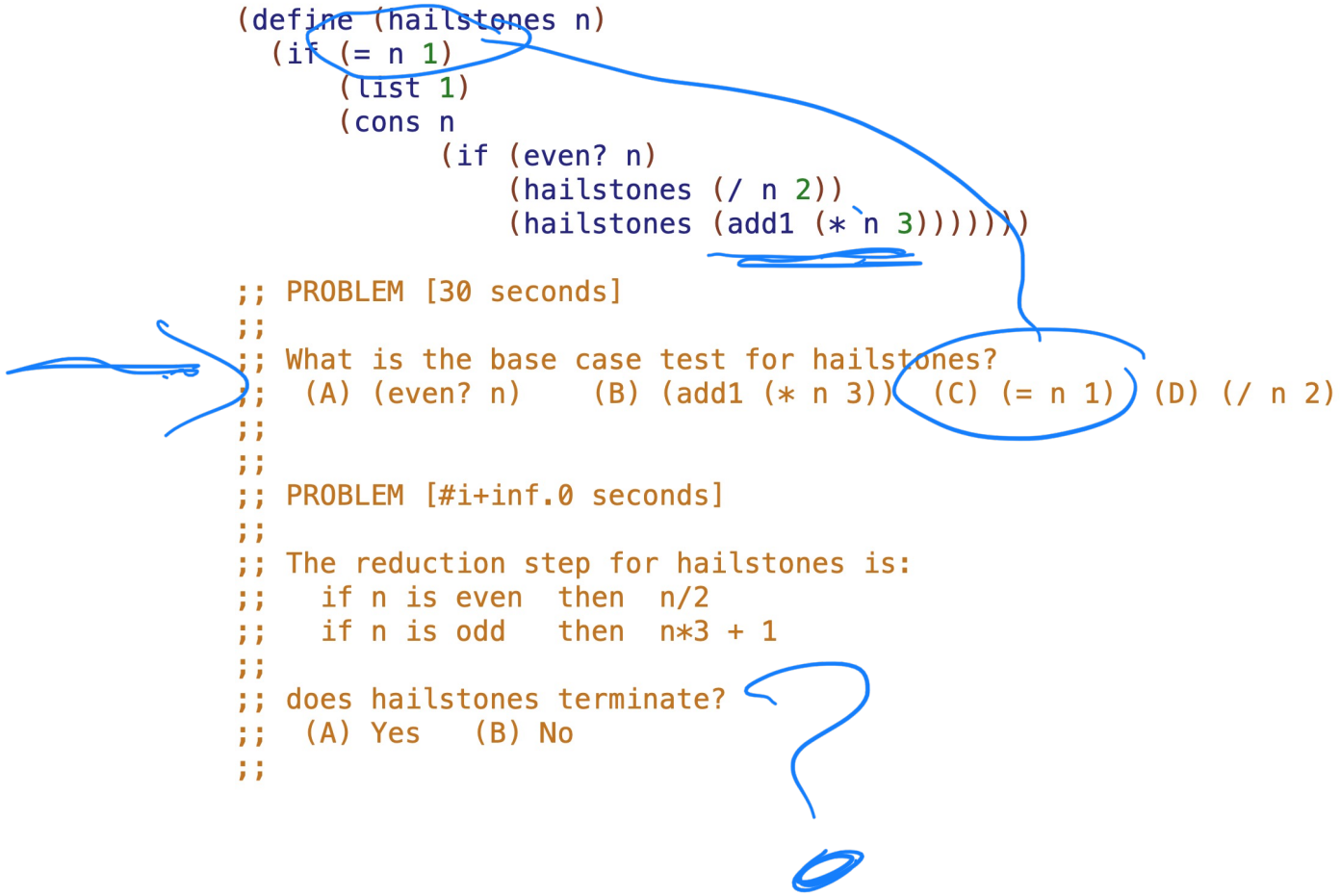
Generative Recursion

The template for generative recursion is:

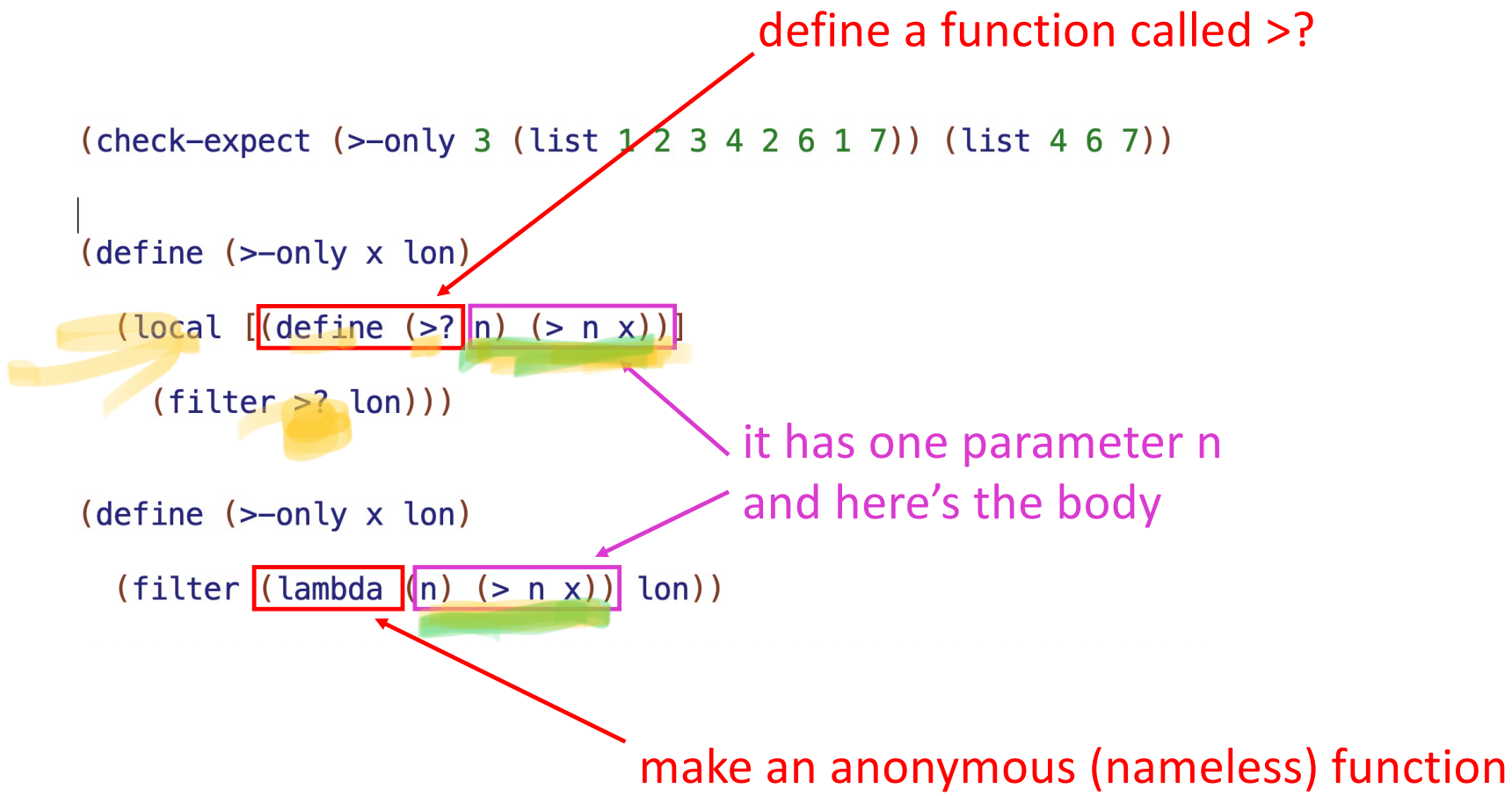
```
(define (genrec-fn d)
  (cond [(trivial? d) (trivial-answer d)]
        [else
         (... d
              (genrec-fn (next-problem d)))]))
```

```
(@htdf silly)
(@signature (listof Number) -> (listof Number))
;; a silly little function on lists of numbers
(check-expect (silly empty) empty)
(check-expect (silly (list 8 4 12 9 10 1 2 3 4))
              (list 8 4 1))

(define (silly lon)
  (if (empty? lon)
      empty
      (local [(define f (first lon))
              (define (<f? x) (< x f))]
        (cons f
              (silly (filter <f? (rest lon)))))))

;;
;; PROBLEM [40 seconds]
;;
;; What is the trivial case test for silly?
;;  (A) filter    (B) (rest lon)  (C) lon is empty  (D) 1
;;
;;
;; PROBLEM [40 seconds]
;;
;; What is the reduction step for silly?
;;  (A) (rest lon)    (B) (first lon)  (C) elements of (rest lon) < (first lon)
;;
;; PROBLEM [30 seconds]
;;
;; Does silly always terminate?  Meaning for any list we call it with,
;; does it always produce a result?
;;  (A) YES   (B) NO
```

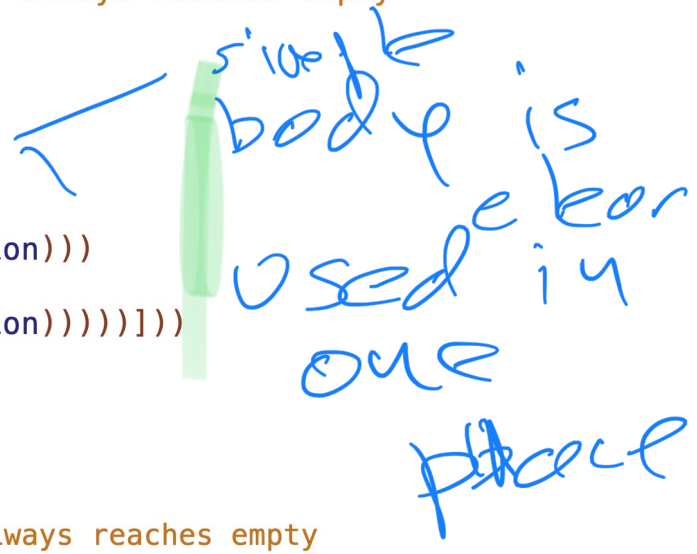
```
(@htdf hailstones)
(@signature Integer -> (listof Integer))
;; produce hailstone sequence for n
;; CONSTRAINT integers in argument and result are >=1
(check-expect (hailstones 1) (list 1))
(check-expect (hailstones 2) (list 2 1))
(check-expect (hailstones 4) (list 4 2 1))
(check-expect (hailstones 5) (list 5 16 8 4 2 1))

(define (hailstones n)
  (if (= n 1)
      (list 1)
      (cons n
            (if (even? n)
                (hailstones (/ n 2))
                (hailstones (add1 (* n 3)))))))

;; PROBLEM [30 seconds]
;;
;; What is the base case test for hailstones?
;;  (A) (even? n)    (B) (add1 (* n 3))   (C) (= n 1)   (D) (/ n 2)
;;
;;
;; PROBLEM [#i+inf.0 seconds]
;;
;; The reduction step for hailstones is:
;;   if n is even  then  n/2
;;   if n is odd   then  n*3 + 1
;;
;; does hailstones terminate?
;;  (A) Yes   (B) No
;;
```



```
(@template-origin genrec use-abstract-fn)

(define (qsort lon)
  ;; Base case: empty
  ;; Reduction: strict subset (at least one element removed) from lon
  ;; Argument: repeated strict subset of list always reaches empty
  (cond [(empty? lon) empty]
        [else
         (local [(define p (first lon))
                 (define (<p? x) (< x p))
                 (define (>p? x) (> x p))]

           (append (qsort (filter <p? (rest lon)))
                   (list p)
                   (qsort (filter >p? (rest lon))))))]))
```

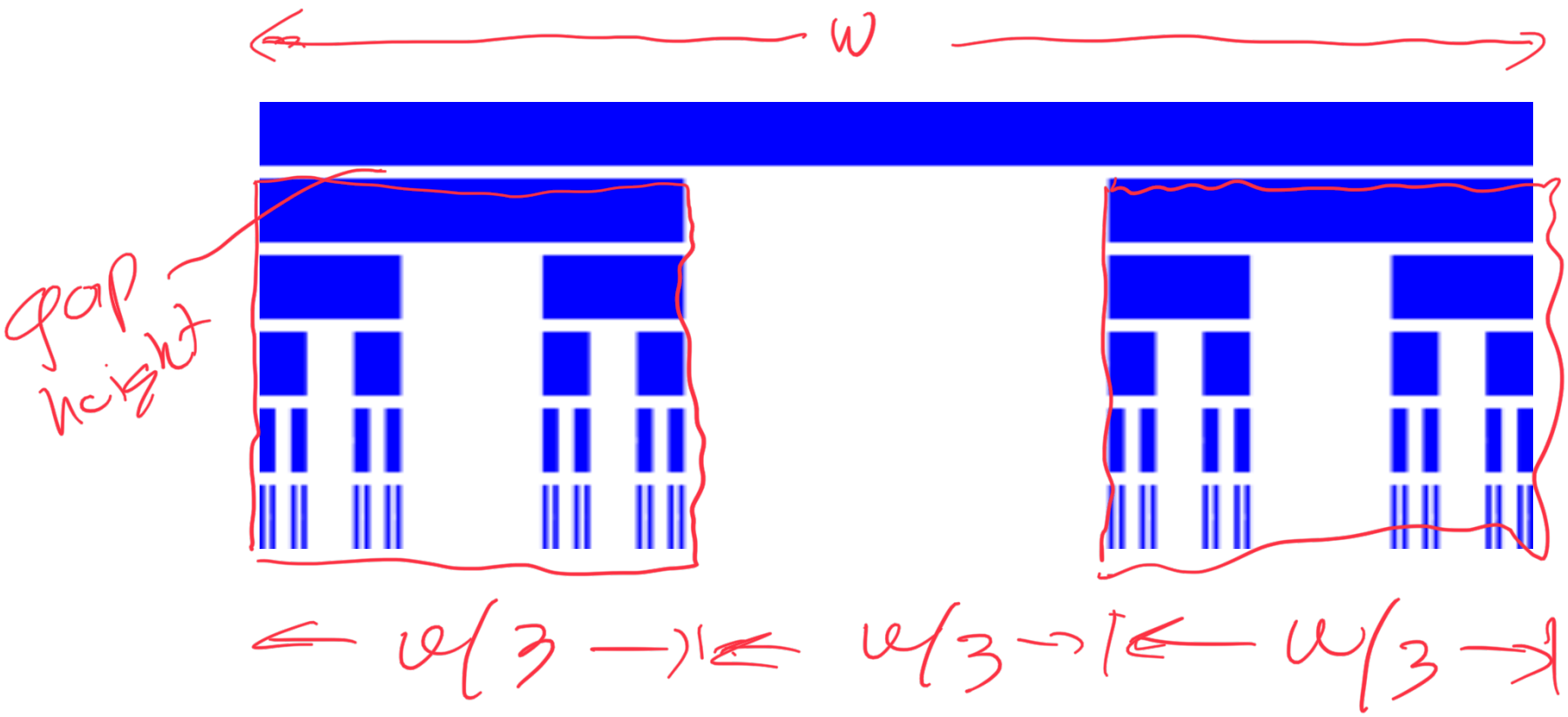

define a function called >?
(check-expect (>-only 3 (list 1 2 3 4 2 6 1 7)) (list 4 6 7))
(define (>-only x lon)
(local [(define (>? n) (> n x))]
(filter >? lon)))
it has one parameter n
and here's the body
(define (>-only x lon)
(filter (lambda (n) (> n x)) lon))
make an anonymous (nameless) function

```
(@template-origin genrec use-abstract-fn)

(define (qsort lon)
  ;; Base case: empty
  ;; Reduction: strict subset (at least one element removed) from lon
  ;; Argument: repeated strict subset of list always reaches empty
  (cond [(empty? lon) empty]
        [else
         (local [(define p (first lon))
                 (define (<p? x) (< x p))
                 (define (>p? x) (> x p))]

           (append (qsort (filter <p? (rest lon)))
                   (list p)
                   (qsort (filter >p? (rest lon))))))]))
```




```
(define (qsort lon)
  ;; Base case: empty
  ;; Reduction: strict subset of lon
  ;; Argument: repeated strict subset of list always reaches empty
  (cond [(empty? lon) empty]
        [else
         (local [(define p (first lon))]
           (append (qsort (filter (lambda (x) (< x p)) (rest lon)))
                   (list p)
                   (qsort (filter (lambda (x) (> x p)) (rest lon))))))]))
```

Center 300



Cantor 3) → 1

12

```
(@template-origin genrec)

(define (cantor w)
  ;; base case:  w <= CUTOFF, (note CUTOFF is > 0)
  ;; reduction:  w / 3
  ;; argument:   repeated division by 3 approaches 0, so will get below CUTOFF

  (cond [(<= w CUTOFF) (rectangle w BAR-HEIGHT "solid" BAR-COLOR)]
        [else
         (local [(define w/3 (/ w 3))
                 (define top (rectangle w BAR-HEIGHT "solid" BAR-COLOR))
                 (define gap (rectangle w GAP-HEIGHT "solid" SPACER-COLOR))
                 (define l&r (cantor w/3))
                 (define spc (rectangle w/3 BAR-HEIGHT "solid" SPACER-COLOR))]
           (above top
                  gap
                  (beside l&r spc l&r)))]))
```

names to vals