CSE 1729:Principles of Programming

Lecture 15: Sorting Part 3

Kaleel Mahmood

Department of Computer Science and Engineering

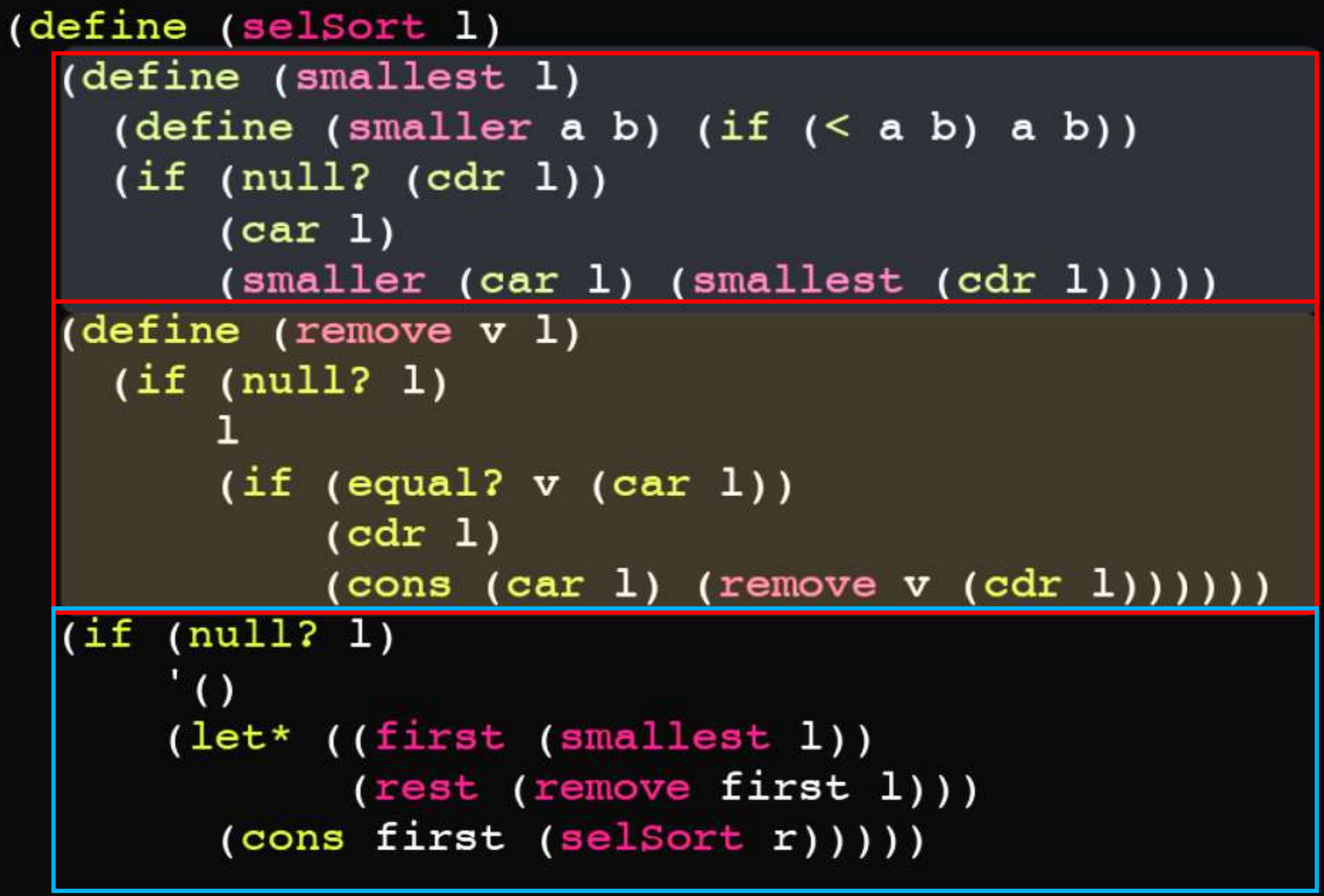
University of Connecticut

Review from before the break (and midterm)

- We had created all the pieces in Scheme to do selection sort:

1. Finding the minimum element in a list.
2. Removing an element from a list.
3. Putting it together: Every time go through the current list, remove the smallest and add it to a sorted list (which starts out as empty).

AND…TO MINIMIZE CLUTTER



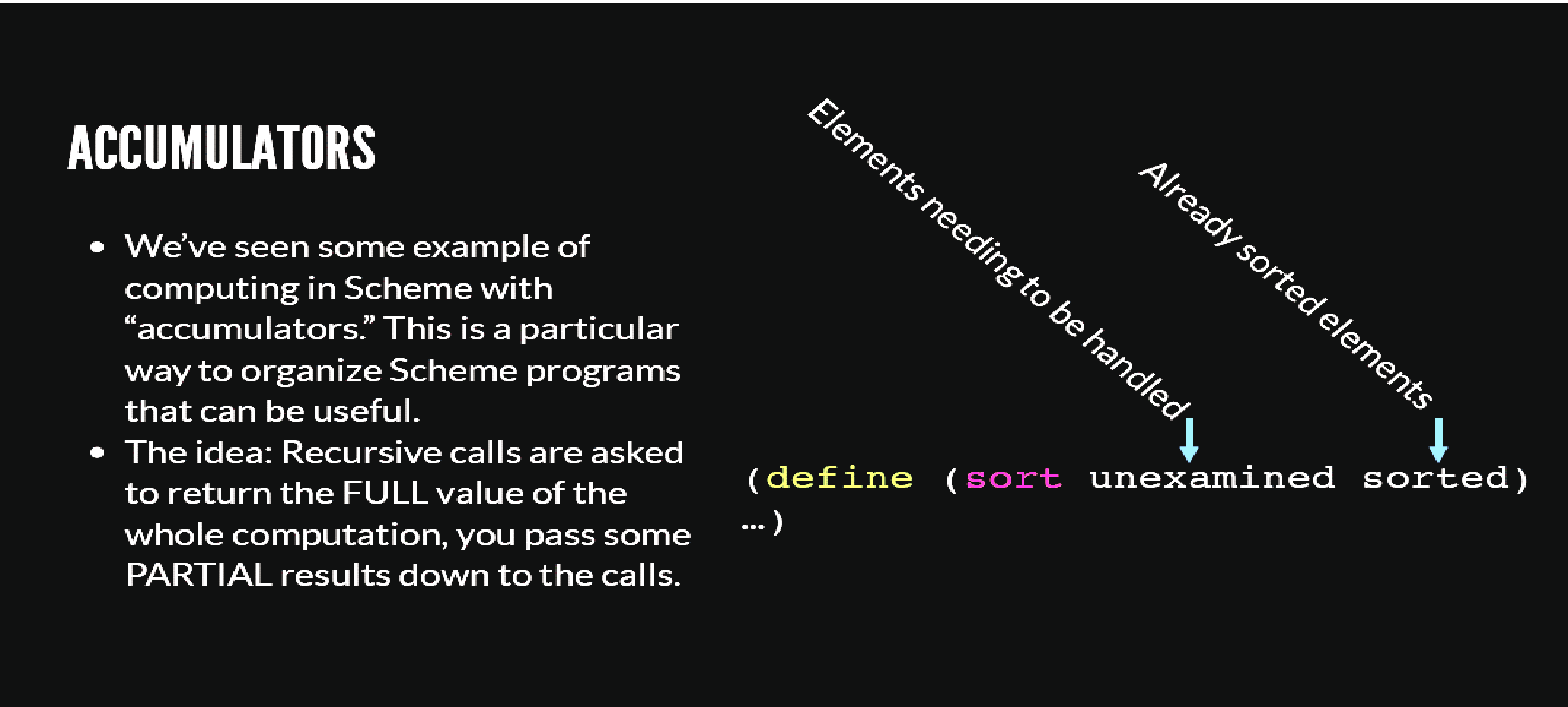
```scheme
(define (selSort l)
  (define (smallest l)
    (define (smaller a b) (if (< a b) a b))
    (if (null? (cdr l))
        (car l)
        (smaller (car l) (smallest (cdr l)))))
  (define (remove v l)
    (if (null? l)
        l
        (if (equal? v (car l))
            (cdr l)
            (cons (car l) (remove v (cdr l))))))
  (if (null? l)
      '()
      (let* ((first (smallest l))
             (rest (remove first l)))
        (cons first (selSort r)))))
```

Code for finding the smallest number in a list

Code for removing an element from a list

Code for doing selection sort using smallest and remove functions

Another way to program Selection Sort:



ACCUMULATORS

- We've seen some example of computing in Scheme with "accumulators." This is a particular way to organize Scheme programs that can be useful.
- The idea: Recursive calls are asked to return the FULL value of the whole computation, you pass some PARTIAL results down to the calls.

```
(define (sort unexamined sorted)
…)
```

How to find the smallest crab using accumulators?

- **The smallest crab problem**: I give you three buckets. A bucket of crabs and two empty buckets. You also get a scale. If you can find the smallest crab I will give you an A in the course. If you don't find the smallest crab I will give you an F.
- The constraints: If you let any crabs escape you also fail. I expect you to return to me 3 buckets at the end. One bucket contains the smallest crab. The other bucket contains all remaining crabs. The last bucket should be empty.
- If you take a crab out of the bucket and leave it on the beach it will run away (then you fail).

How to find the smallest crab using only three buckets?

- Denote the bucket with all the crabs that have not been sorted as the dirty bucket.

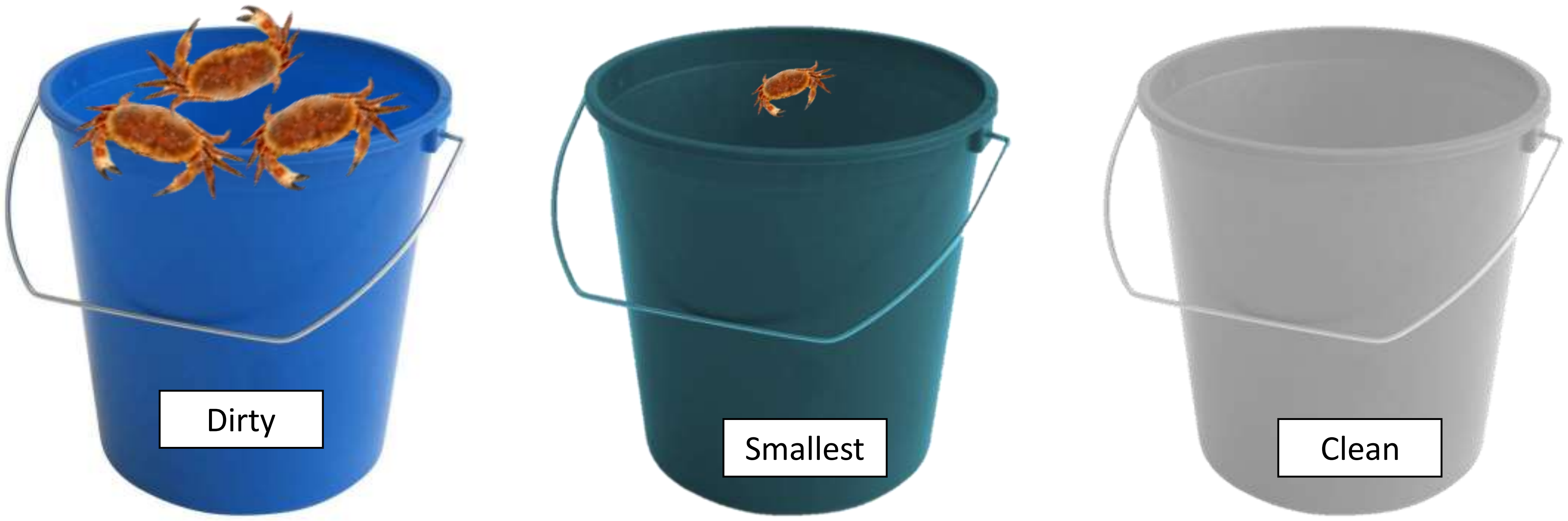
How to find the smallest crab using only three buckets?

- Denote the bucket with the current smallest crab as the “smallest” bucket.

Dirty
Smallest

How to find the smallest crab using only three buckets?

- Denote the bucket with the crabs we have already looked at that are NOT the smallest as the “clean” bucket.



Now consider the logic for finding the smallest crab. There are two cases:

- Case 1: The current crab we are looking at is smaller than the current smallest crab. What do you do?

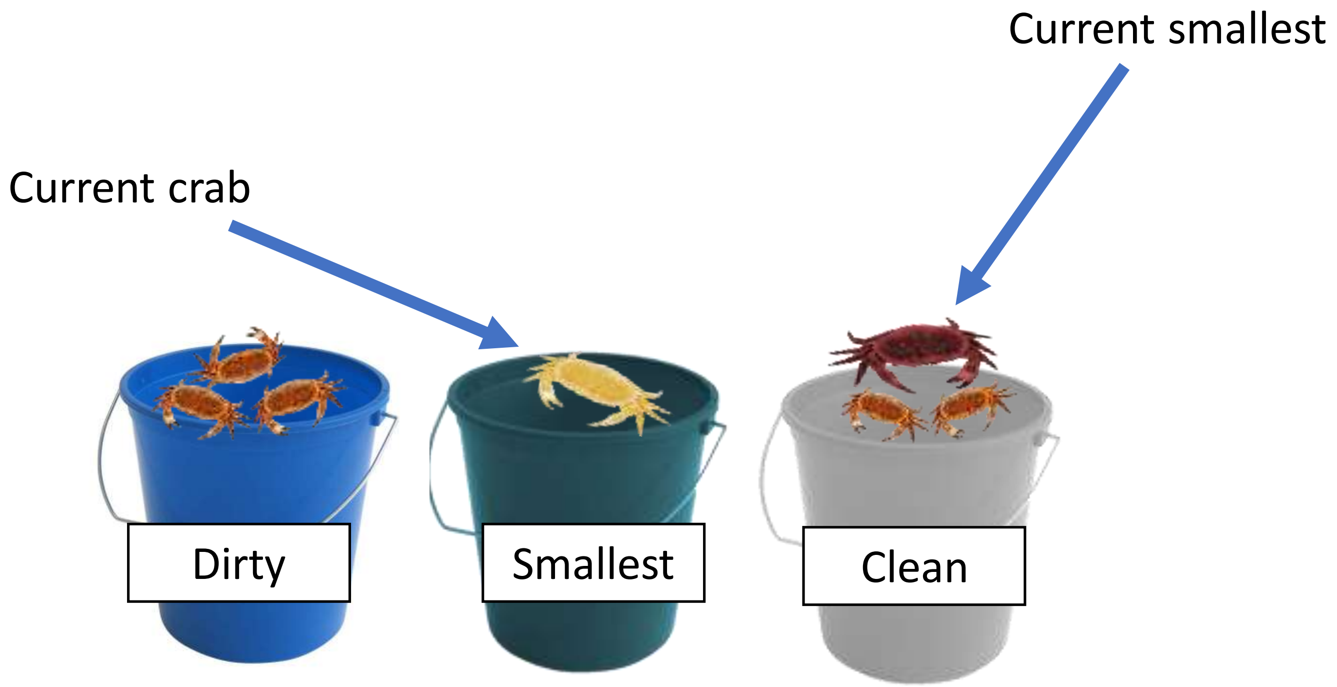
Current crab
Current smallest

Dirty
Smallest
Clean

Now consider the logic for finding the smallest crab. There are two cases:

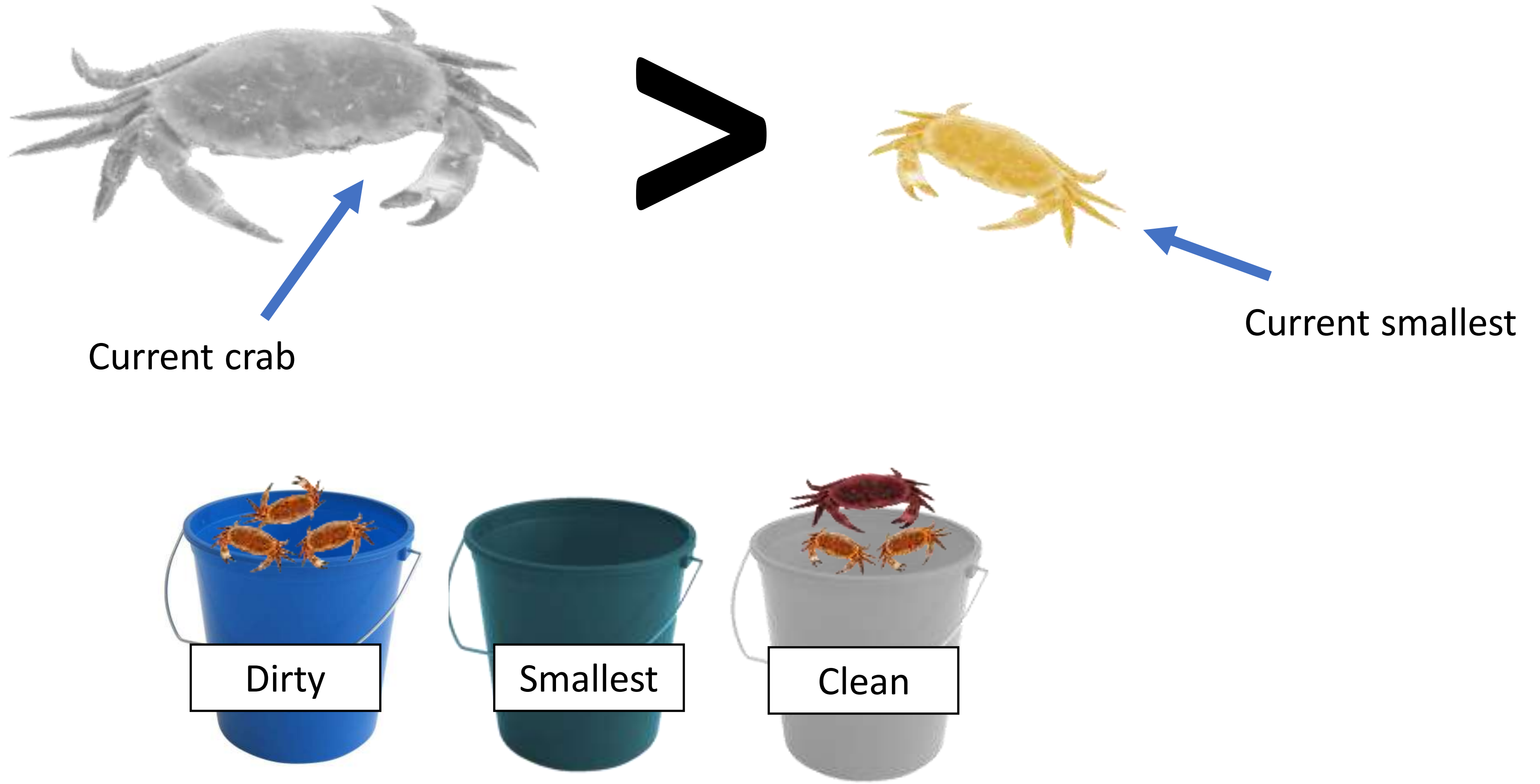
- Case 1: The current crab we are looking at is smaller than the current smallest crab. What do you do? Do two things:

1. Put the current crab in the smallest bucket.
2. Put the current smallest in the clean bucket (it is no longer the smallest)



Now consider the logic for finding the smallest crab. There are two cases:

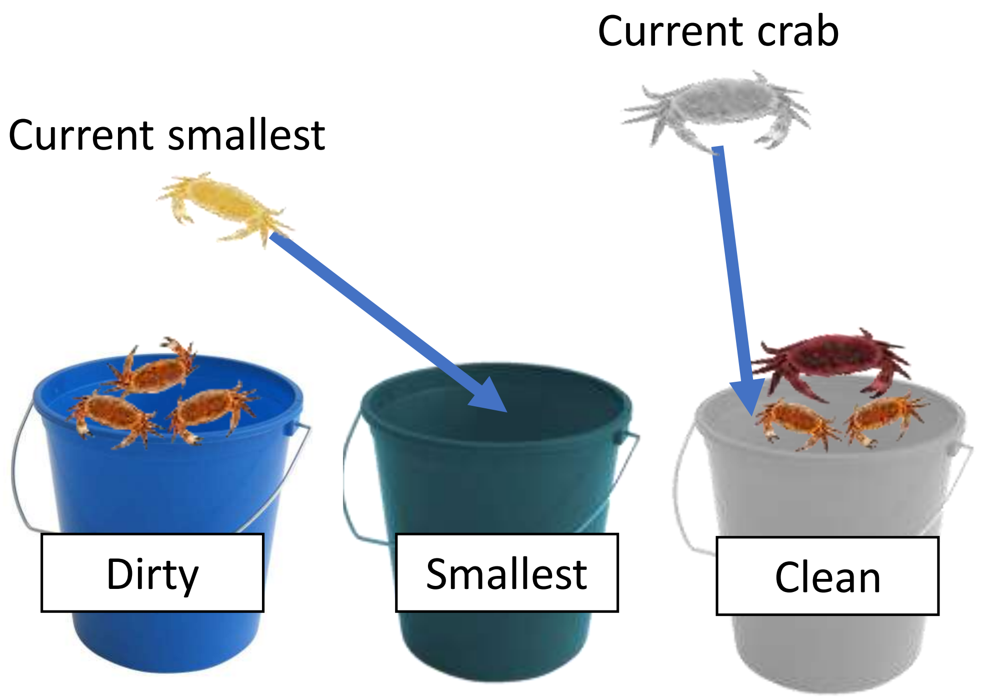
- Case 2: The current crab we are looking at is BIGGER than the current smallest crab. What do you do?



Now consider the logic for finding the smallest crab. There are two cases:

- Case 2: The current crab we are looking at is BIGGER than the current smallest crab. What do you do? Do two things:

1. Put the current crab in the CLEAN bucket.
2. Put the current smallest back in the smallest bucket (it is still the smallest).



A SOLUTION USING ACCUMULATORS

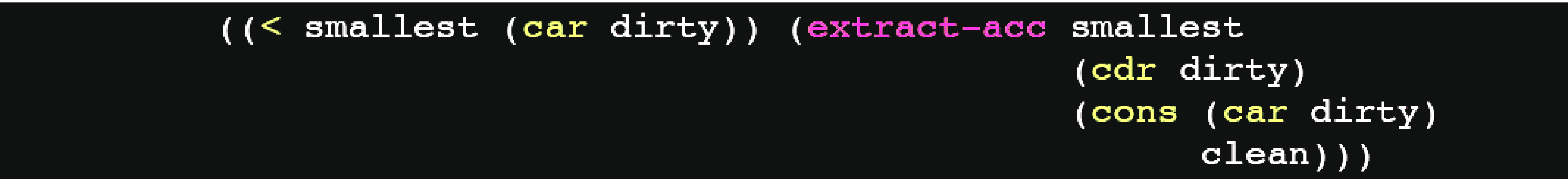
Smallest so far

Unexamined elts

Examined elts

```scheme
(define (alt-extract elements)
  (define (extract-acc smallest dirty clean)
    (cond ((null? dirty) (make-pair smallest clean))
          ((< smallest (car dirty)) (extract-acc smallest
                                                 (cdr dirty)
                                                 (cons (car dirty)
                                                       clean)))
          (else (extract-acc (car dirty)
                             (cdr dirty)
                             (cons smallest clean)))))
  (extract-acc (car elements) (cdr elements) '()))
```

Let's break down the code a little bit…

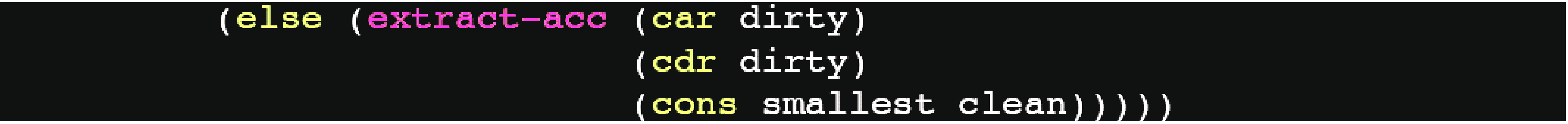


```
((< smallest (car dirty)) (extract-acc smallest
                                        (cdr dirty)
                                        (cons (car dirty)
                                              clean)))
```

Condition: the smallest element (think crab) in the list is SMALLER than the element you just pulled out of the dirty list (dirty bucket).

Variable Assignments using pseudocode
"-" means minus
"=>" Means assign

smallest => smallest (no change)
clean list=> clean list + first element of dirty list
dirty list => current dirty list – the first element in the dirty list



```
        (else (extract-acc (car dirty)
                           (cdr dirty)
                           (cons smallest clean)))))
```

Condition: the smallest element (think crab) in the list is BIGGER than the element you just pulled out of the dirty list (dirty bucket).

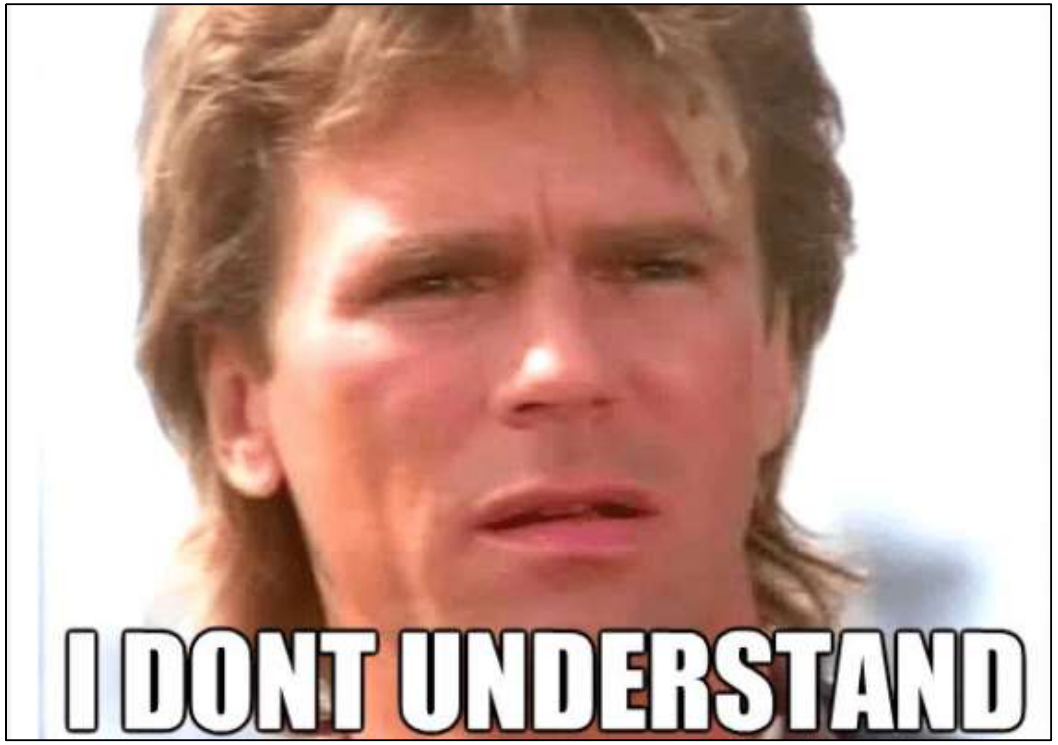
Variable Assignments using pseudocode
“-” means minus
“=>” Means assign

clean list=> clean list + the previous smallest element
smallest => first element of dirty list
dirty list => current dirty list – the first element in the dirty list

WHAT’S THE DIFFERENCE?

- In our original solution,
  - “partial problems” are passed as parameters;
  - “partial solutions” are passed back as values.
- In our accumulator solution,
  - “partial solutions” are passed as parameters;
  - complete solutions are passed back as values.
- Trace a short evaluation!
- Both of these are good techniques to keep in mind;
  - some problems can be more elegantly factored one way or the other.

But why would we program using an accumulator?



- It is faster? No.
- Is it an easy design pattern to understand and re-use? Maybe.
- Is it tail recursive? Yes!

WHAT ABOUT ANOTHER ORDERING?

- For instance....
  - Get the sorted list in decreasing order!
- Wish
  - Do not duplicate all the code.

- Pass a function that embodies the order we wish to use.
- Examples

```
(selSort (lambda (a b) (< a b))
         (list 3 6 1 0 7 4 2 8 9 5 12))

(selSort (lambda (a b) (> a b))
         (list 3 6 1 0 7 4 2 8 9 5 12))
```

Output:

```
(0 1 2 3 4 5 6 7 8 9 12)
(12 9 8 7 6 5 4 3 2 1 0)
```

SELECTION SORT WITH AN EXTERNALIZED ORDERING

```
(define (selSort before? l)
  (define (smallest l)
    (define (choose a b) (if (before? a b) a b))
    (if (null? (cdr l))
        (car l)
        (choose (car l) (smallest (cdr l)))))
  (define (remove v l)
    (if (null? l)
        l
        (if (equal? v (car l))
            (cdr l)
            (cons (car l) (remove v (cdr l))))))
  (if (null? l)
      '()
      (let* ((f (smallest l))
             (r (remove f l)))
        (cons f (selSort r)))))
```

That's it! No other changes needed!

Yet.... `before?` is used from choose not from `selSort`.
How does this work?

CLOSURE

- It’s all about the environments!
  - When entering `selSort`, the environment has a binding for `before?`
  - When defining `smallest`, scheme uses the current environment
    - Therefore `before?` is *still in the environment.*
  - When defining choose scheme evaluates `before?`; and picks up its definition from the current environment!

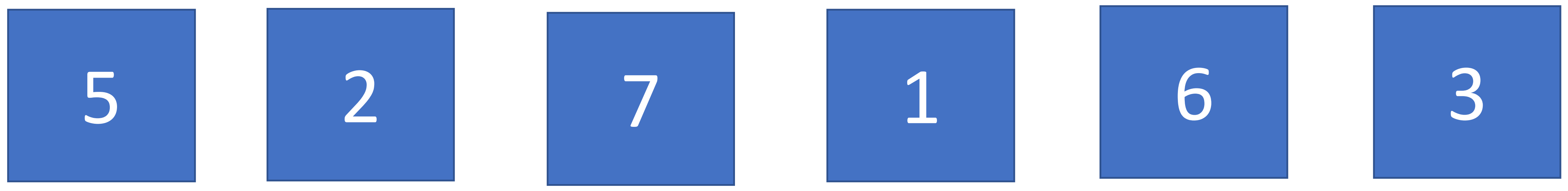
The definition of choose has captured a reference to `before?`

Another Sorting Algorithm: Quicksort

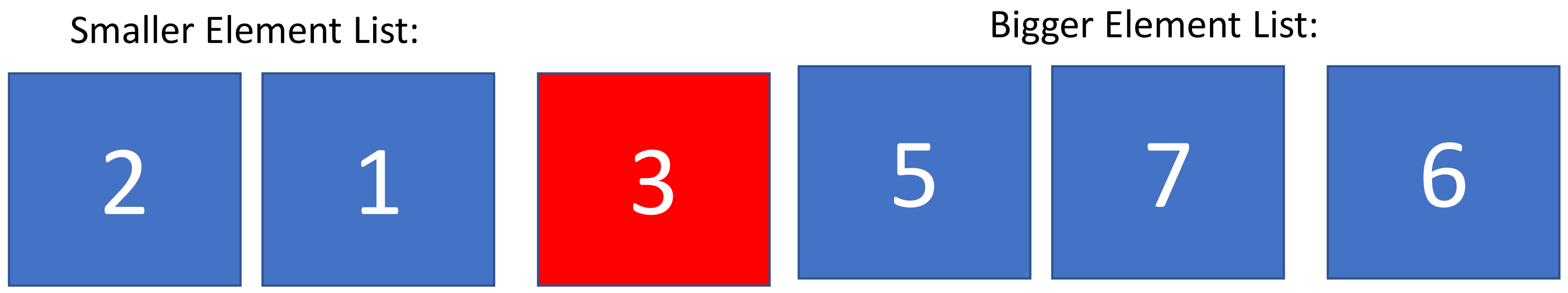
Basic idea:

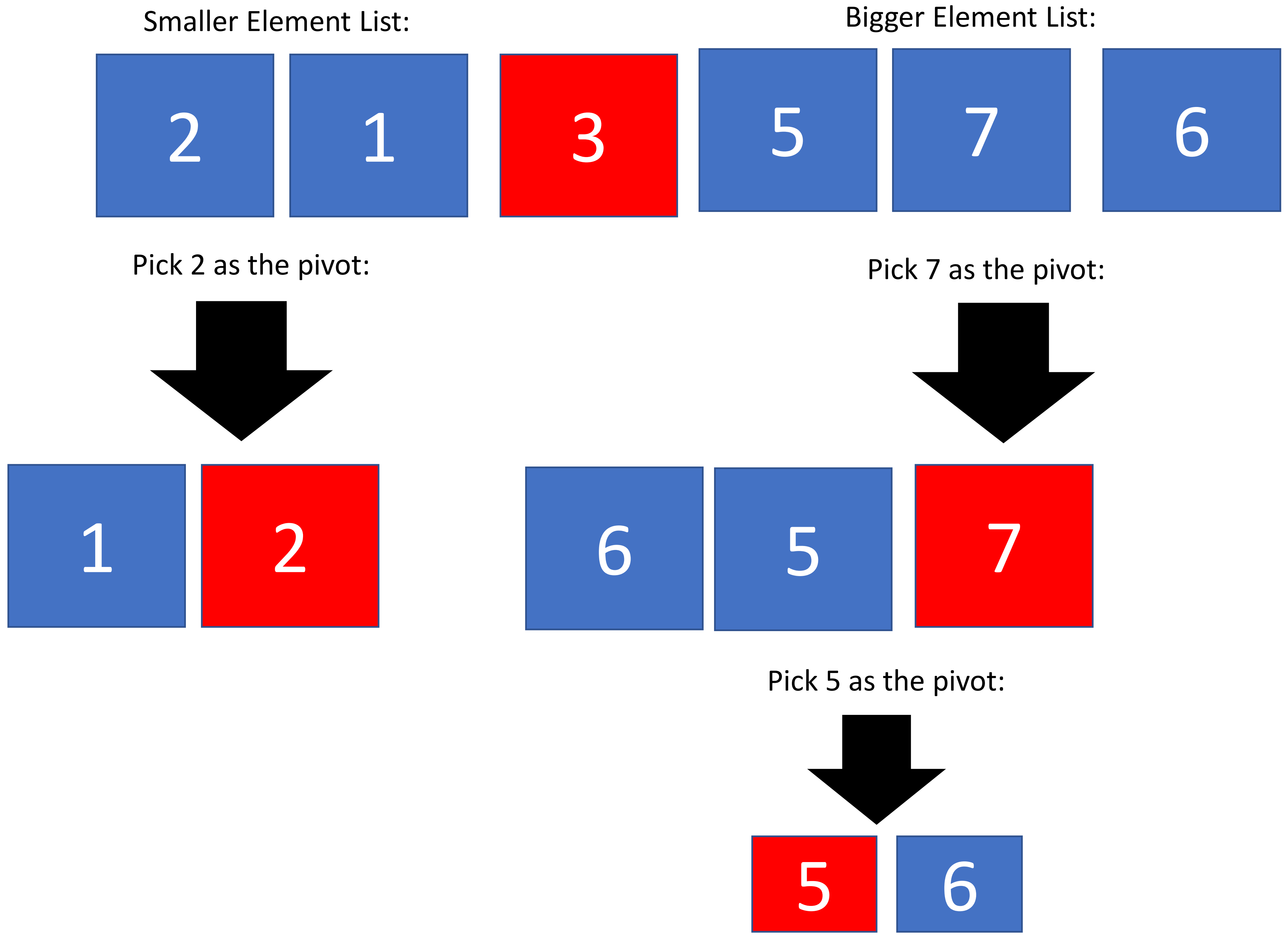
1. Choose a random element in the list.
2. Use this element as a pivot and divide the list into two parts.
3. Repeat step 1 with each of the two parts of the list.
4. Continue steps 1 through 3 until you reach lists that ONLY have one element (base case).

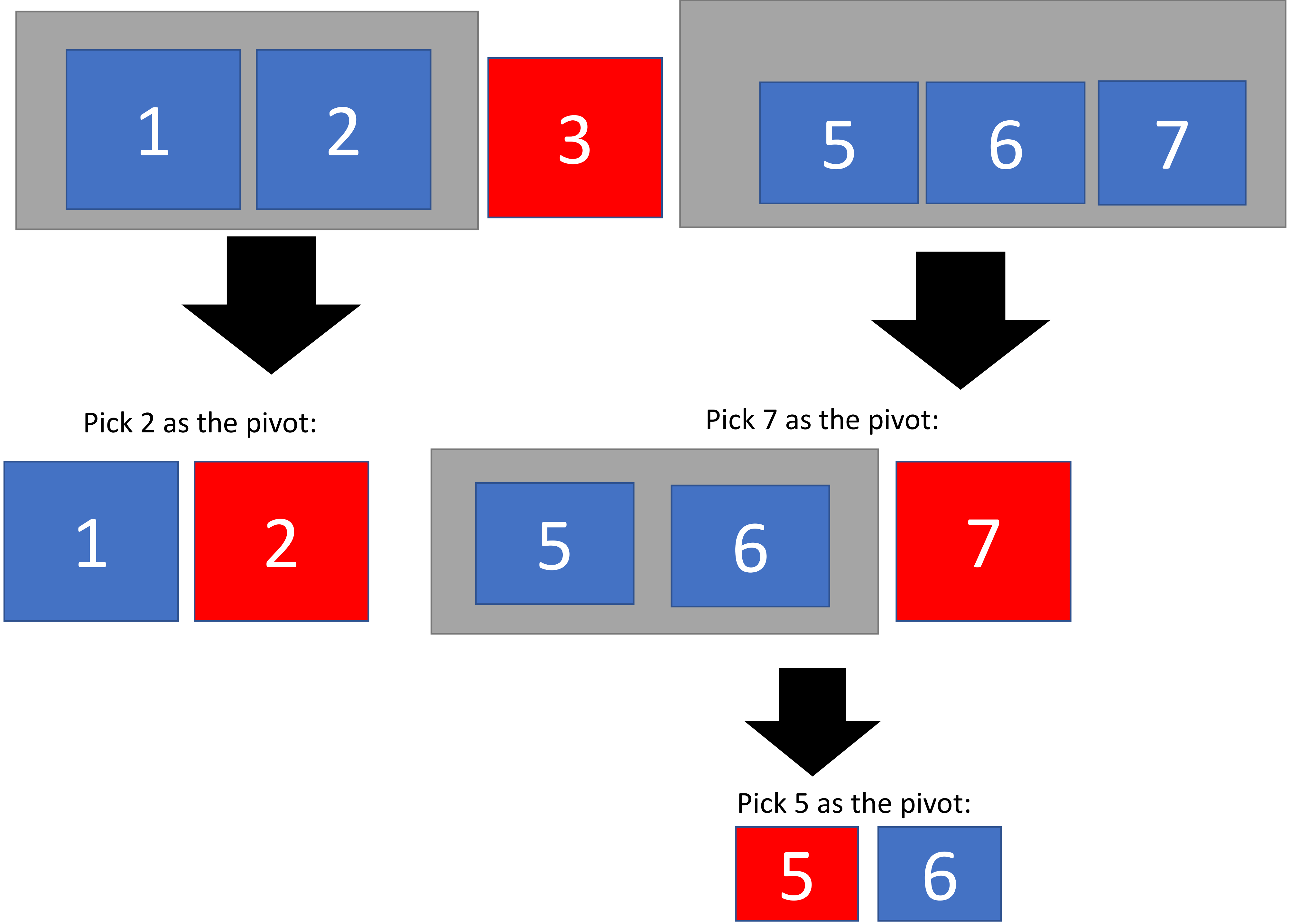
Pictorial Example



Randomly pick 3 as the pivot:




Smaller Element List:
Bigger Element List:
2
1
3
5
7
6
Pick 2 as the pivot:
Pick 7 as the pivot:
1
2
6
5
7
Pick 5 as the pivot:
5
6


1
2
3
5
6
7
Pick 2 as the pivot:
1
2
Pick 7 as the pivot:
5
6
7
Pick 5 as the pivot:
5
6

KEY INGREDIENTS

- Partitioning
  - Use a *pivoting* element
  - Throw values smaller than *pivot* on left
  - Throw values larger than *pivot* on right
- Sorting
  - Pick a pivot
  - Partition
  - Sort partitions recursively (What is the base case?)
  - Combine answers

Step 1: Partitioning based on pivot

```
(define (partition l pivot left right)
  (cond ((null? l)                (make-pair left right))
        ((< (car l) pivot)        (partition  (cdr l)
                                               pivot
                                              (cons (car l) left)
                                               right))
        (else                     (partition (cdr l)
                                             pivot
                                             left
                                             (cons (car l) right)))))
```

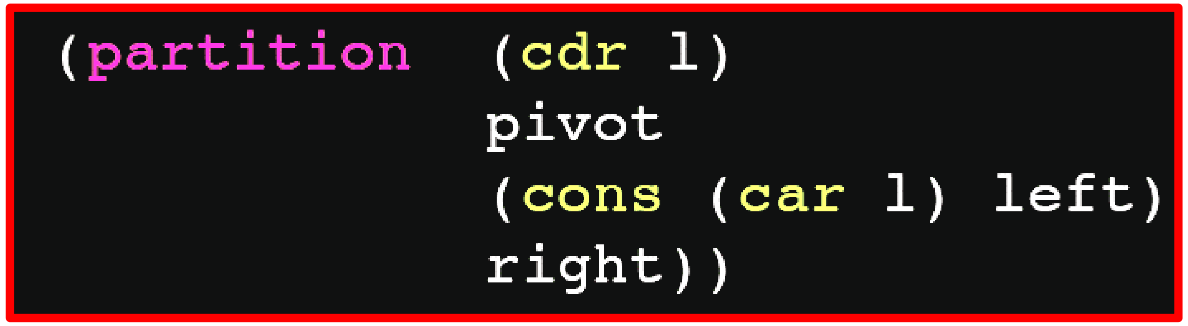
```
(cond ((null? l)          (make-pair left right))
```

Base case: Nothing left in the list so just put the left and right part of the list together and return!

Step 1: Partitioning based on pivot

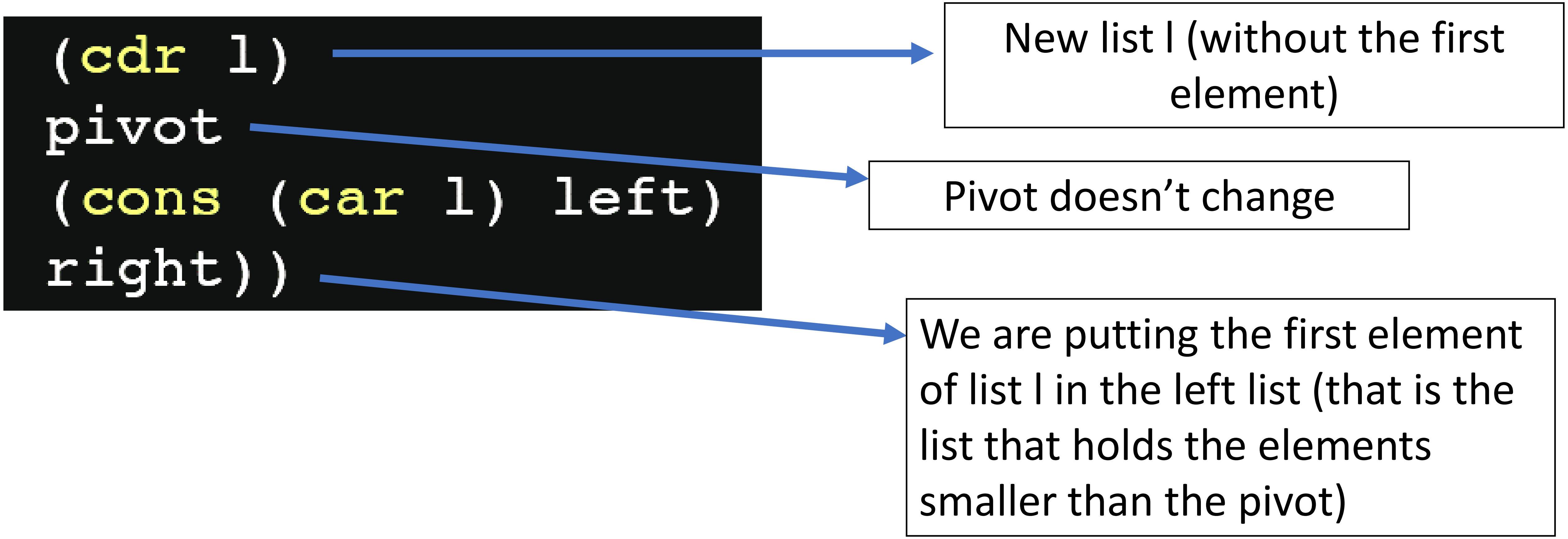
```
(define (partition l pivot left right)
[redacted]
        ((< (car l) pivot)  (partition  (cdr l)
                                        pivot
                                        (cons (car l) left)
                                        right))
[redacted]
```

Case where the first element in the list is SMALLER than the pivot.
See next slide for what is going on in the input function call…



```
(partition  (cdr l)
            pivot
            (cons (car l) left)
            right))
```

Breakdown of the function call:



```
(cdr l)
pivot
(cons (car l) left)
right))
```

New list l (without the first element)

Pivot doesn't change

We are putting the first element of list l in the left list (that is the list that holds the elements smaller than the pivot)

Step 1: Partitioning based on pivot

```scheme
(define (partition l pivot left right)
  (cond ((null? l)              (make-pair left right))
        ((< (car l) pivot)  (partition  (cdr l)
                                         pivot
                                         (cons (car l) left)
                                         right))
        (else               (partition (cdr l)
                                        pivot
                                        left
                                        (cons (car l) right)))))
```

Hopefully the last case is obvious now. Just to be safe we can go through it as well…

```
(partition (cdr l)
           pivot
           left
           (cons (car l) right)))))
```

Breakdown of the function call:

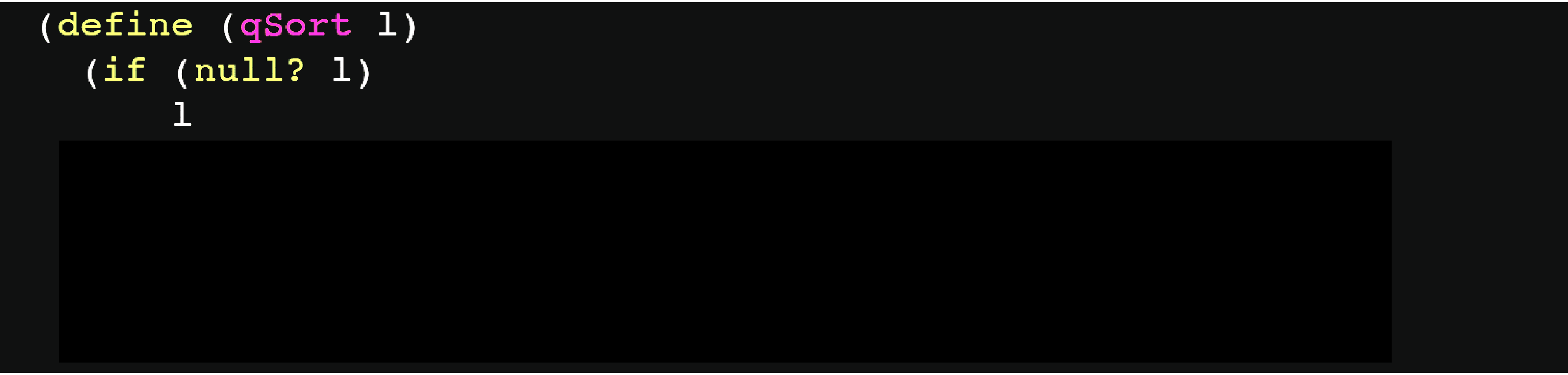
```
(cdr l)
pivot
left
(cons (car l) right)))))
```

New list l (without the first element)

Pivot doesn't change

We are putting the first element of list l in the right list (that is the list that holds the elements larger than the pivot)

Step 2: Putting it all together

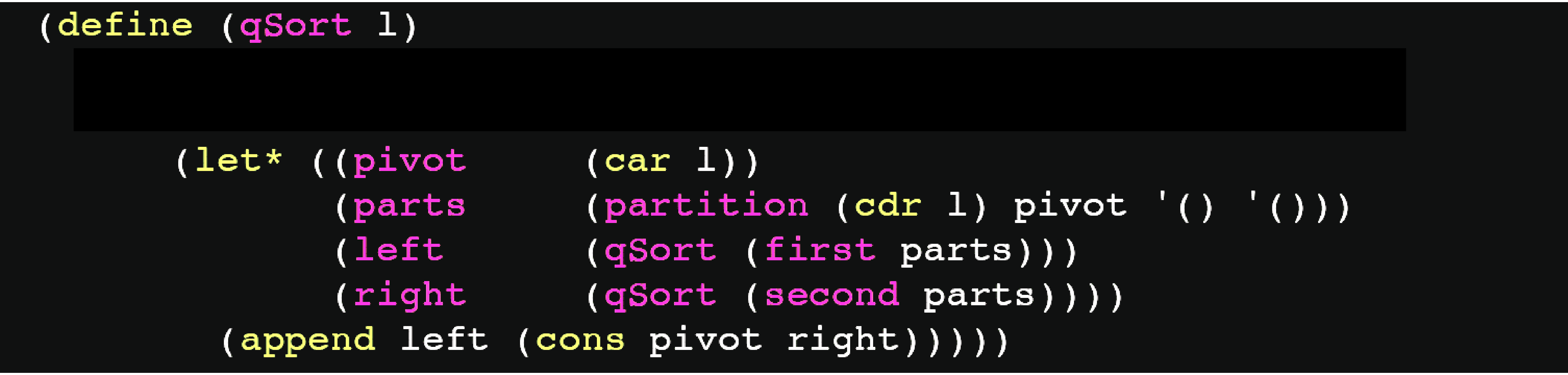


```
(define (qSort l)
  (if (null? l)
      l
```

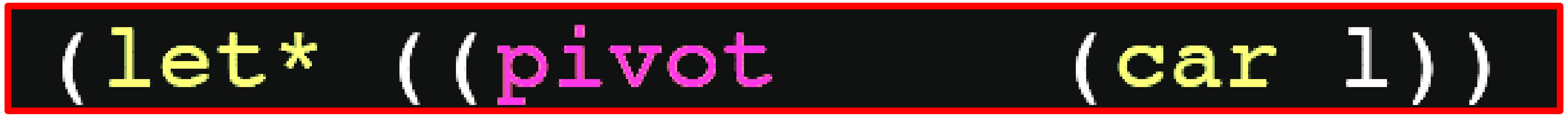
[redacted]

If the list is null don’t do anything….

Step 2: Putting it all together



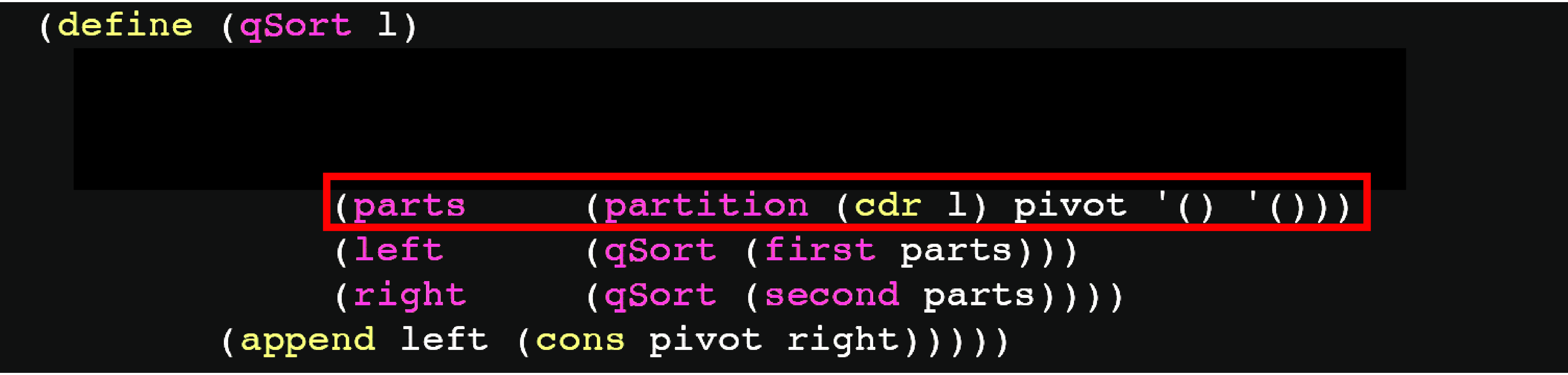
```
(define (qSort l)
[redacted]
      (let* ((pivot      (car l))
             (parts      (partition (cdr l) pivot '() '()))
             (left       (qSort (first parts)))
             (right      (qSort (second parts))))
        (append left (cons pivot right)))))
```



```
(let* ((pivot      (car l))
```

Select the first element in the unsorted list to be the pivot.

Step 2: Putting it all together



```
(define (qSort l)
    [redacted]
                  (parts       (partition (cdr l) pivot '() '()))
                  (left        (qSort (first parts)))
                  (right       (qSort (second parts))))
            (append left (cons pivot right)))))
```

```
(parts       (partition (cdr l) pivot '() '()))
```

Call the partition function.
The starting list is cdr(l) that means all elements except for the first element in the list. Why?

Because the first element is being used as the pivot.

Step 2: Putting it all together

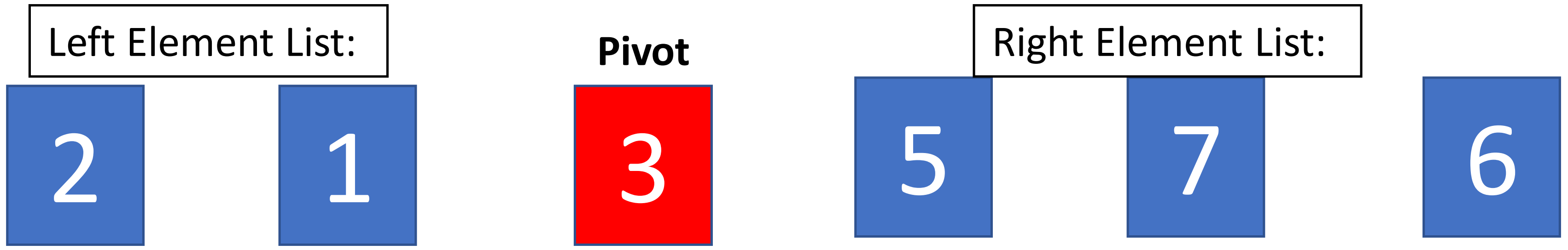
```
(define (qSort l)
   [redacted]
             (left        (qSort (first parts)))
             (right       (qSort (second parts))))
        (append left (cons pivot right)))))
```

After we've gotten the left and right lists from the pivot what is next? Well we need to sort each part. So call qSort on each of the lists.

Step 2: Putting it all together

```
(define (qSort l)
[redacted]
        (append left (cons pivot right)))))
```

Last step: when all elements have been placed in a left or right list, return the entire list with the left (smaller elements), the pivot and then the right list (larger elements)

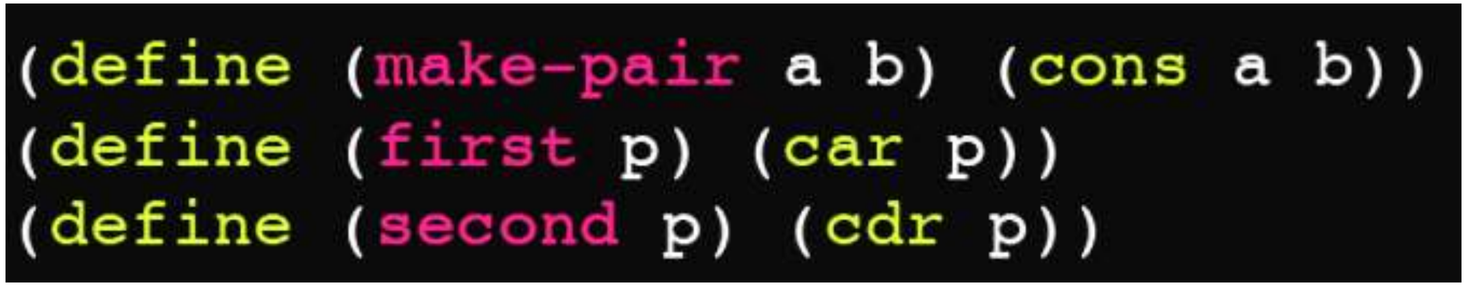


Big Picture: All the code for QuickSort

```
(define (partition l pivot left right)
  (cond ((null? l)            (make-pair left right))
        ((< (car l) pivot)   (partition  (cdr l)
                                          pivot
                                          (cons (car l) left)
                                          right))
        (else                (partition (cdr l)
                                         pivot
                                         left
                                         (cons (car l) right)))))
```

```
(define (qSort l)
  (if (null? l)
      l
      (let* ((pivot     (car l))
             (parts     (partition (cdr l) pivot '() '()))
             (left      (qSort (first parts)))
             (right     (qSort (second parts))))
        (append left (cons pivot right)))))
```

Extra Functions We Previously Defined



```
(define (make-pair a b) (cons a b))
(define (first p) (car p))
(define (second p) (cdr p))
```

Main Lecture Concept: How can you understand complicated codes like this?

1. First choose a simple but illustrative example. E.g. in QuickSort using a list with only two elements might not be a good way to understand the code.
2. In your example make sure you know what the CORRECT solution should be (don’t guess). This correct solution can also be called the ground truth.
3. Go through the code line by line with pen and paper. Write down variable names and function calls as you step through the code.
4. Trace back up through the code when you hit the base case.

Figure Sources

- https://www.liveabout.com/thmb/PvE1TXn3D7lN8Xj4FZDQSj_CnVI=/801x398/filters:no_upscale():max_bytes(150000):strip_icc()/confusedmacgyvermeme-5ab8392eae9ab8003778609b.PNG
- https://eandt.theiet.org/media/10468/cute-little-crab.jpg?anchor=center&mode=crop&width=640&height=480&rnd=132247695260000000
- https://images.immediate.co.uk/production/volatile/sites/30/2020/02/Crab-body-7f9ae78.jpg?quality=90&resize=768,574