CSE1729: Introduction to Programming

Structured Data in SCHEME: Pairs and lists

Greg Johnson

Our story thus far...

- ...has focused on two "data-types:" numbers and functions. (In fact, numeric data types are rather more complicated than you might think at first: recall the difference between 4 and 4.0.)
- However, we often want to construct and manipulate more complicated *structured* data objects:
    - pairs of objects,
    - lists of objects,
    - trees, graphs, expressions, ...

Pairs

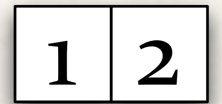
- Scheme has built-in support for *pairs* of objects. To maintain pairs, we require:
  - **A method for producing a pair from two objects:** In SCHEME, this is the `cons` function. It takes two arguments and returns a pair containing the two values.
  - **A method of extracting the first (resp. second) object from a pair:** In SCHEME, these are two chimerically named functions: `car` and `cdr`. Given a pair p, `(car p)` returns the first object in p; `(cdr p)` returns the second.

Examples; notation

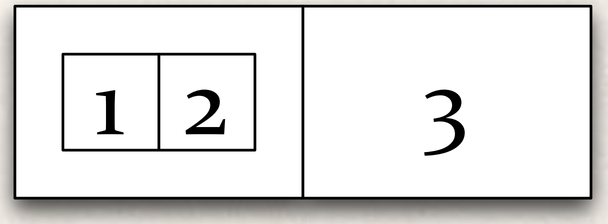
```
> (cons 1 2)
(1 . 2)
> (define p (cons 1 2))
> (car p)
1
> (cdr p)
2
> (define q (cons p 3))
> (car q)
(1 . 2)
> (cdr q)
3
> (car (car q))
1
> (cdr (car q))
2
>
```

- Note that the interpreter denotes the pair containing the two objects a and b as: `(a . b)`.
- Note that a coordinate of a pair can be...*another pair*! A natural diagram to represent this situation:

`(cons 1 2)`



`(cons (cons 1 2) 3)`



A complex number datatype

- Recall that a complex number can be written a + bi, where i is the a square root of -1. To express a complex, we need to maintain two numbers---the real part and the complex part. We'll use SCHEME pairs to represent complexes. The first coordinate will hold the real part; the second coordinate will hold the complex part. Thus:

- construct a new complex number

```
(define (make-complex a b) (cons a b))
```

- Extract the real part of a complex

```
(define (real-coeff c) (car c))
```

- Extract the imaginary part of a complex

```
(define (imag-coeff c) (cdr c))
```

Operating on complexes

- Adding complexes:

```
(define (add-complex c d)
  (make-complex (+ (real-coeff c) (real-coeff d))
                (+ (imag-coeff c) (imag-coeff d))))
```

- Multiplying complexes $(a_1 + b_1 i)(a_2 + b_2 i) = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + b_1 a_2)i$. Thus:

```
(define (mult-complex c d)
  (make-complex (- (* (real-coeff c) (real-coeff d))
                   (* (imag-coeff c) (imag-coeff d)))
                (+ (* (real-coeff c) (imag-coeff d))
                   (* (imag-coeff c) (real-coeff d)))))
```

Other basic operations

- Conjugate
```
(define (conjugate c)
  (make-complex (real-coeff c)
                (* -1 (imag-coeff c))))
```

- Modulus (length): two natural definitions:

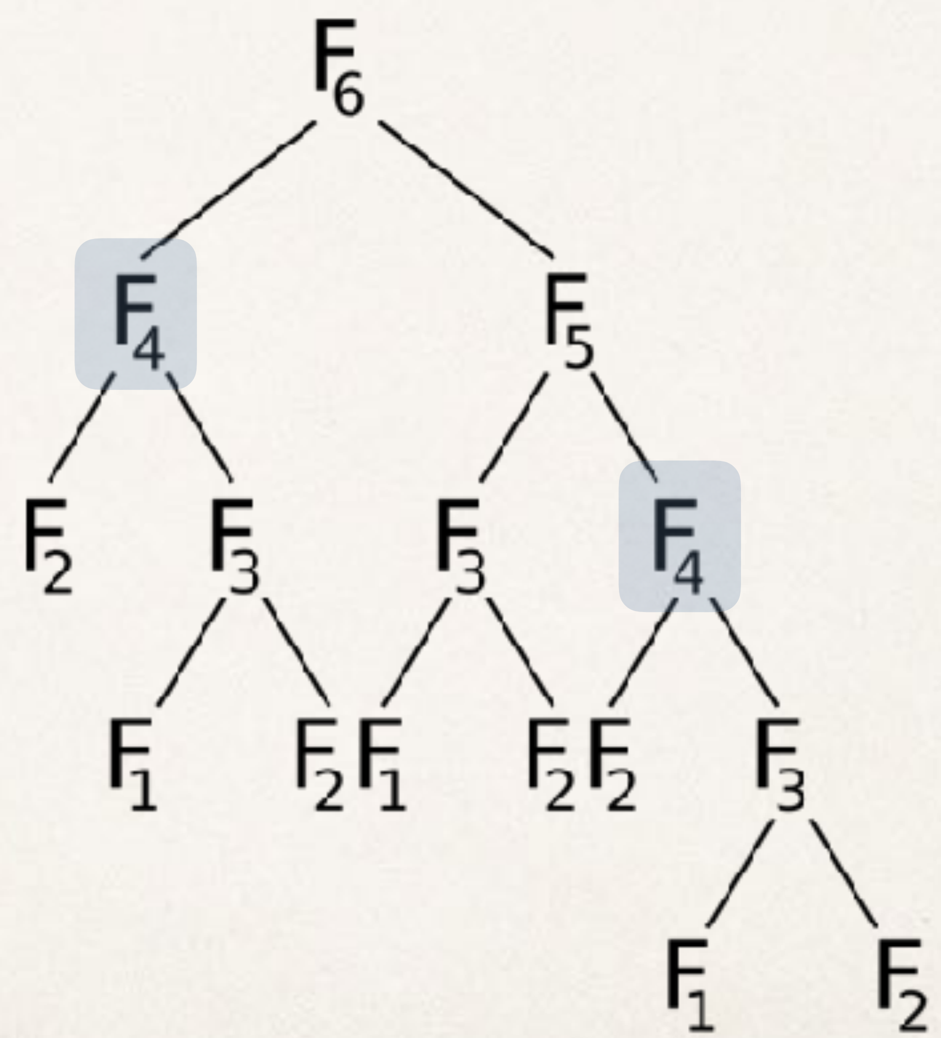
```
(define (modulus c)
  (sqrt (real-coeff (mult-complex c (conjugate c)))))
```

or

```
(define (modulus-alt c)
  (define (square x) (* x x))
  (sqrt (+ (square (real-coeff c))
           (square (imag-coeff c)))))
```

Recall our program for computing the Fibonacci numbers…

- **Problem**. It's a nice, declarative program, but…it inefficient! It does the same work over and over…
- See how (f 4) is called twice? The entire computation is done twice.
- If only there was a better way…



Fast Fibonacci numbers, reinvented with pairs

- We noted earlier that the naive definition of the Fibonacci numbers is costly, requiring a number of a recursive calls roughly equal to the number we are computing. In particular, is it not possible to compute F_{100} by this method on a modern computer.

- Note, in contrast, that it is easy to compute the pair (F_{n+1}, F_n) from the pair (F_n, F_{n-1}) (since $F_{n+1} = F_n + F_{n-1}$).

- This idea can be turned in to a fast definition for the Fibonacci sequence: the idea is for `(fib-pair n)` to return the pair (F_n, F_{n-1}).

Fast Fibonacci numbers

- Note that the n^{th} pair can be computed from the n-1^{st} pair in a straightforward way. Then the n^{th} Fibonacci number can be computed with approximately n additions!

```
(define (fast-fib n)
  (define (fib-pair n)
    (if (= n 1)
        (cons 1 0)
        (let ((prev-pair (fib-pair (- n 1))))
          (cons (+ (car prev-pair)
                   (cdr prev-pair))
                (car prev-pair)))))
  (car (fib-pair n)))
```

Returns the n^{th} Fib pair

The pair

Rational numbers are pairs

- A natural way to maintain a rational number is as a pair

```
(define (make-rat a b)
  (cons a b))

(define (denom r) (cdr r))
(define (numer r) (car r))
```

- Then, to multiply two rationals:

```
(define (mult-rat r s)
  (make-rat (* (numer r) (numer s))
            (* (denom r) (denom s))))
```

Rational addition, reduced form

- To add, we implement the familiar rule:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

- Thus:

```
(define (add-rat r s)
  (make-rat (+ (* (numer r) (denom s))
               (* (numer s) (denom r)))
            (* (denom r) (denom s))))
```

- Note that this implementation does not reduce fractions into reduced form.

Reducing a fraction

- Note that
$$\frac{a}{b} = \frac{a/\alpha}{b/\alpha} \quad \text{if } \alpha \text{ evenly divides } a \text{ and } b$$
- And hence we can always reduce a fraction by the rule:
$$\frac{a}{b} \leadsto \frac{a/\gcd(a,b)}{b/\gcd(a,b)}$$
- We could make a simplify function, or just redefine make-rat, so that all rationals are automatically in reduced form:

```
(define (make-rat a b)
  (let ((d (gcd a b)))
    (cons (/ a d)  (/ b d))))
```

Examples

- Using this new, automatically reducing package:

```
> (define r (make-rat 2 6))
> r
(1 . 3)
> (define s (make-rat 6 15))
> s
(2 . 5)
> (add-rat r s)
(11 . 15)
>
```

2/6 is reduced to 1/3

6/15 is reduced to 2/5

1/3 + 2/5 = 11/15

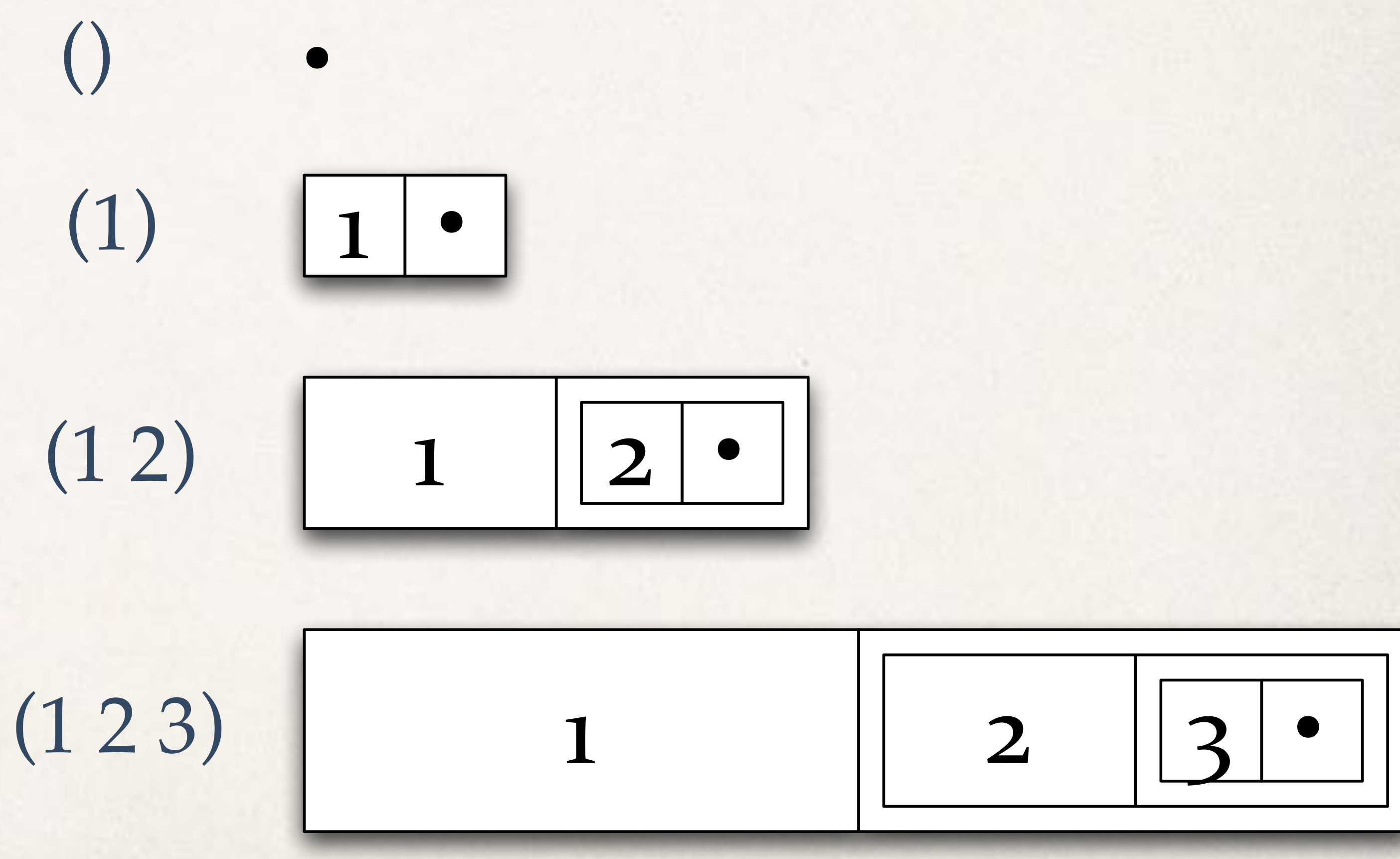
Lists…so important that SCHEME's big sister is named after them

- A *list* is an extremely flexible data structure that maintains an ordered list of objects, for example: *Ceres, Pluto, Makemake, Haumea, Eris,* a list of 5 extrasolar planets.
- SCHEME implements lists **in terms of the pair structure** you have already met. However, pairs have only 2 slots, so we need a mechanism for using pairs to represent lists of arbitrary length.
- Roughly, SCHEME uses the following recursive convention: the list of k objects $a_1,\ldots, a_k$ is represented as a pair where…
  - The first element of the pair is the first element of the list a_1.
  - The second element of the list is…*a list containing the rest of the elements*.

Building up lists with pairs

- To be more precise: A *list* is either
  - the *empty list*, or
  - a *pair*, whose first coordinate is *the first element of the list*, and whose second coordinate is *a list containing the remainder of the elements.*
- Note: *the second element of the pair must be a list.*

- For example, if • denotes the empty list, then...



Some lists: (), (1), (1 2), (1 2 3)

A general list; SCHEME notation

- Thus, a list has the form:

Pair

first element	list of remaining elements

- Since lists are used so frequently, SCHEME provides special notation for them:

```
()
(1)
(1 2)
```

stands for

empty list

```
(1 . ())
(1 . (2 . ()))
```

Note: In SCHEME, **lists are always terminated with the empty list**.

If this looks familiar...

- ...that's good!
- Indeed, you have already been using SCHEME lists.
- SCHEME programs (and expressions) are lists!
- The details...

Quotation; entering lists in the Scheme interpreter

- Recall the SCHEME evaluation rule for compound (list!) objects.

- This means that the natural way to enter a list doesn't work: SCHEME wants to apply evaluation:

```
> ()
. #%app: missing procedure expression; probably originally (), which is an illegal empty application
in: (#%app)
> (1 2)
. . procedure application: expected procedure, given: 1; arguments were: 2
```

- SCHEME provides the `(quote <expr>)` (or `‘<expr>`) form, which evaluates to `<expr>` without further evaluation:

```
> (quote ())
()
> (quote (1 . ()))
(1)
> (quote (1))
(1)
> ‘(1)
(1)
```

Note how SCHEME denotes these identical structures

`‘<expr>` is shorthand for `(quote <expr>)`

Examples; list construction

- It takes some practice to manipulate Scheme lists: the important thing to remember is that if `enemies` is a nonempty list, then `(car enemies)` is the first element of the list and `(cdr enemies)` is the list of all elements after the first.

- Some examples:

```
> (cons 1 2)
(1 . 2)
> (cons 1 '())
(1)
> (cons 1 '(2))
(1 2)
> (cons 1 (cons 2 '()))
(1 2)
> (car '(1 2))
1
> (cdr '(1 2))
(2)
```

A pair

A list

A list is a pair!

Elements of lists can be pairs, functions, other lists, …

- For convenience, SCHEME provides a list constructor function: `list`.
- Note that you can construct lists of arbitrary objects.

```
> (list 1 2 3)
(1 2 3)
> (list (list 1 2) (list 3 4))
((1 2) (3 4))
> (list (cons 1 2) (list 3 4))
((1 . 2) (3 4))
> (list 1 (cons 2 3) (list 4 5))
(1 (2 . 3) (4 5))
> (list 1 2 '())
(1 2 ())
> (list)
()
```

List processing: Handle the first elements and, then,...handle the rest

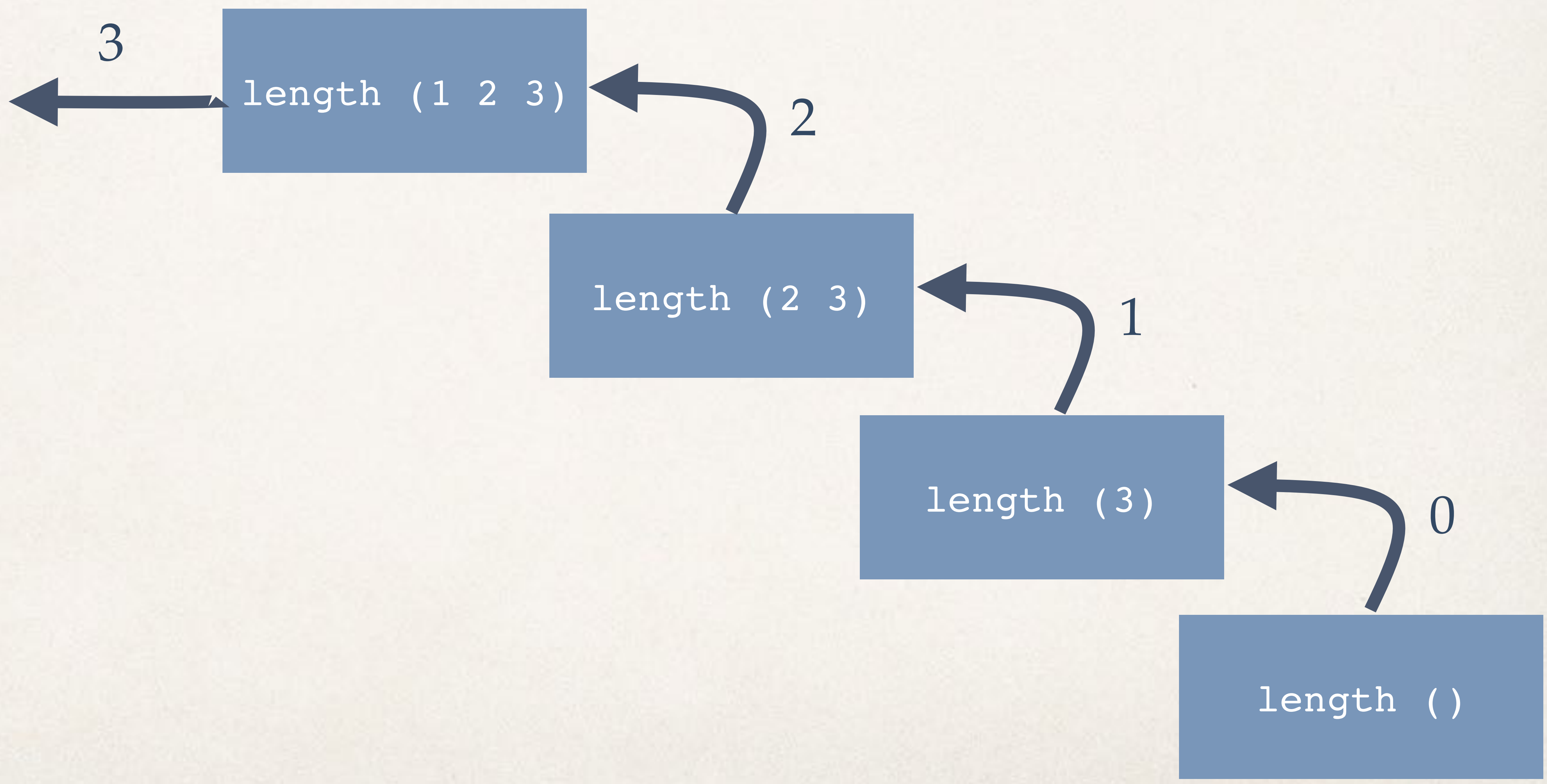
- `(null? x)` returns #t is **x** is the empty list.
- list processing: handle the first element (the `car`) and, then, handle the remaining list (the cdr). Notice that these have different "types."
- Computing the length, for example...

```
(define (nlength xyz)
  (if (null? xyz)
      0
      (+ 1 (nlength (cdr xyz)))))
```

Then...

```
> (nlength '(1 2 3))
3
> (nlength '())
0
> (nlength '((1 2) (3 4)))
2
```

The recursive call structure of a call to `length`



Another example: Summing the numbers of a list

- Adding the elements of a list:

```
(define (sum-list list)
  (if (null? list)
      0
      (+ (car list)
         (sum-list (cdr list)))))
```

- Then...

```
> (sum-list '())
0
> (sum-list '(1 3 5 7))
16
```

Hey, these are great but...not all elements are created equal

- If `list` is a list, it is easy to get to the first element: `(car list)`. The last element, however, takes more work to find. This is an inherent feature (and, sometimes, shortcoming) of this "data structure."

```
(define (last-element l)
  (if (null? (cdr l))
      (car l)
      (last-element (cdr l))))

> (last-element '(5 4 3 2 1))
1
```

Append: Place one list after another.

- Basic operation on lists: place one after the other:

Append `(11 12 13)` to `(1 2 3)` → `(1 2 3 11 12 13)`

- It's easy:

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1)
            (append (cdr list1) list2))))
```

Then...

```
>(append '(1 2 3) '(13 14 15))
(1 2 3 13 14 15)
```

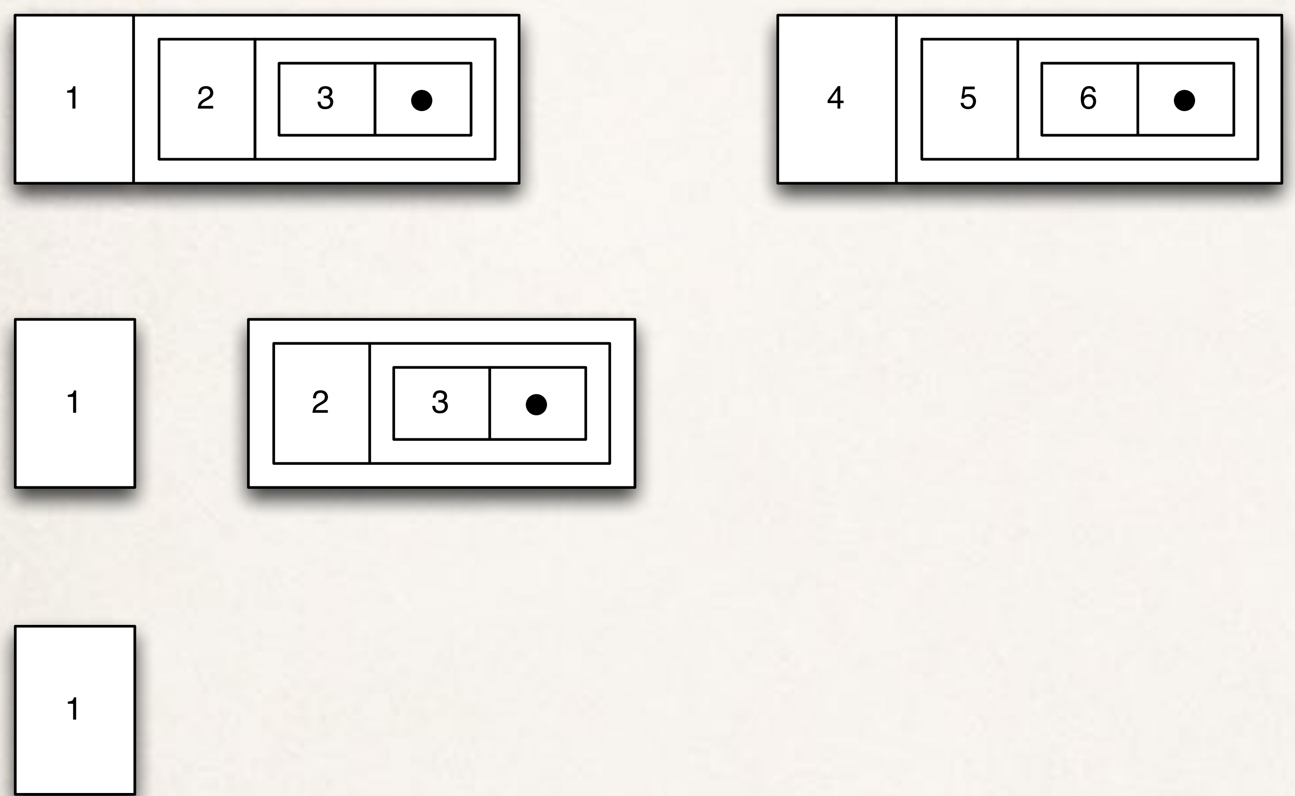
How long does this take?

- A good measure of the “time taken” by a `Scheme` function (without looping constructs, which we will discuss later) is simply the number of recursive calls it generates.

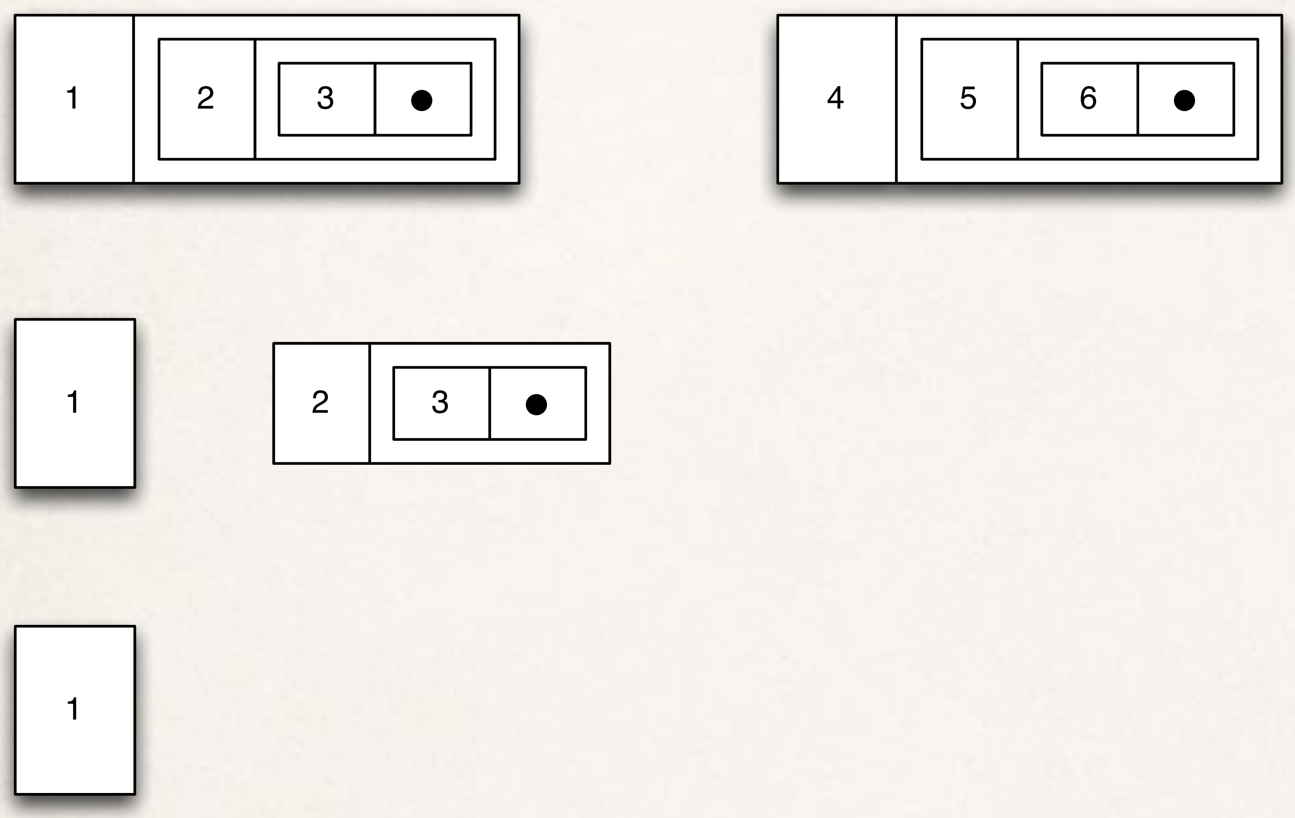
- `(append list1 list2)` involves a total of length(`lista`) recursive calls. (Why? I needs to find the end of the list.)

(append (list 1 2 3) (list 4 5 6))

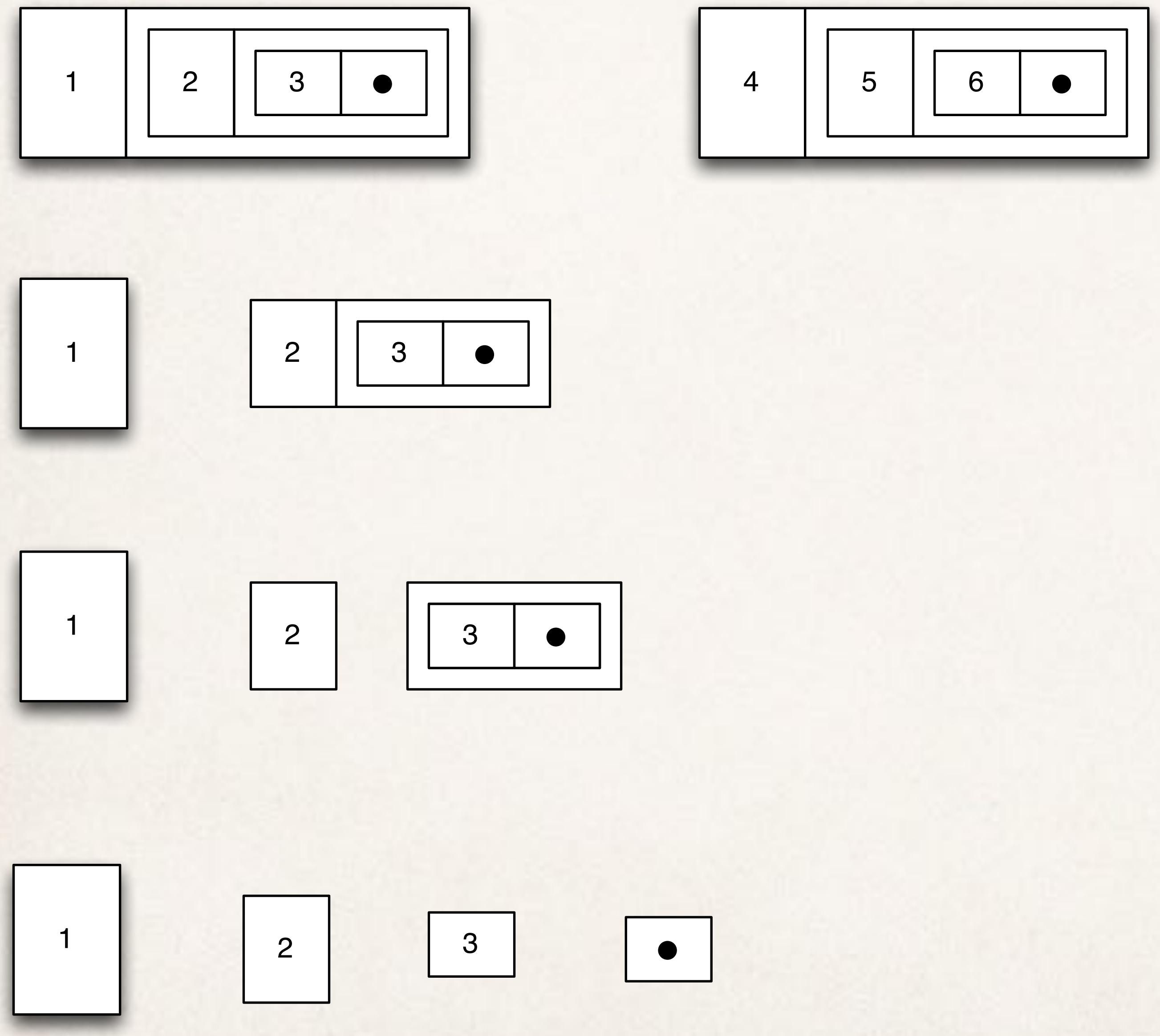
(append (list 1 2 3) (list 4 5 6))



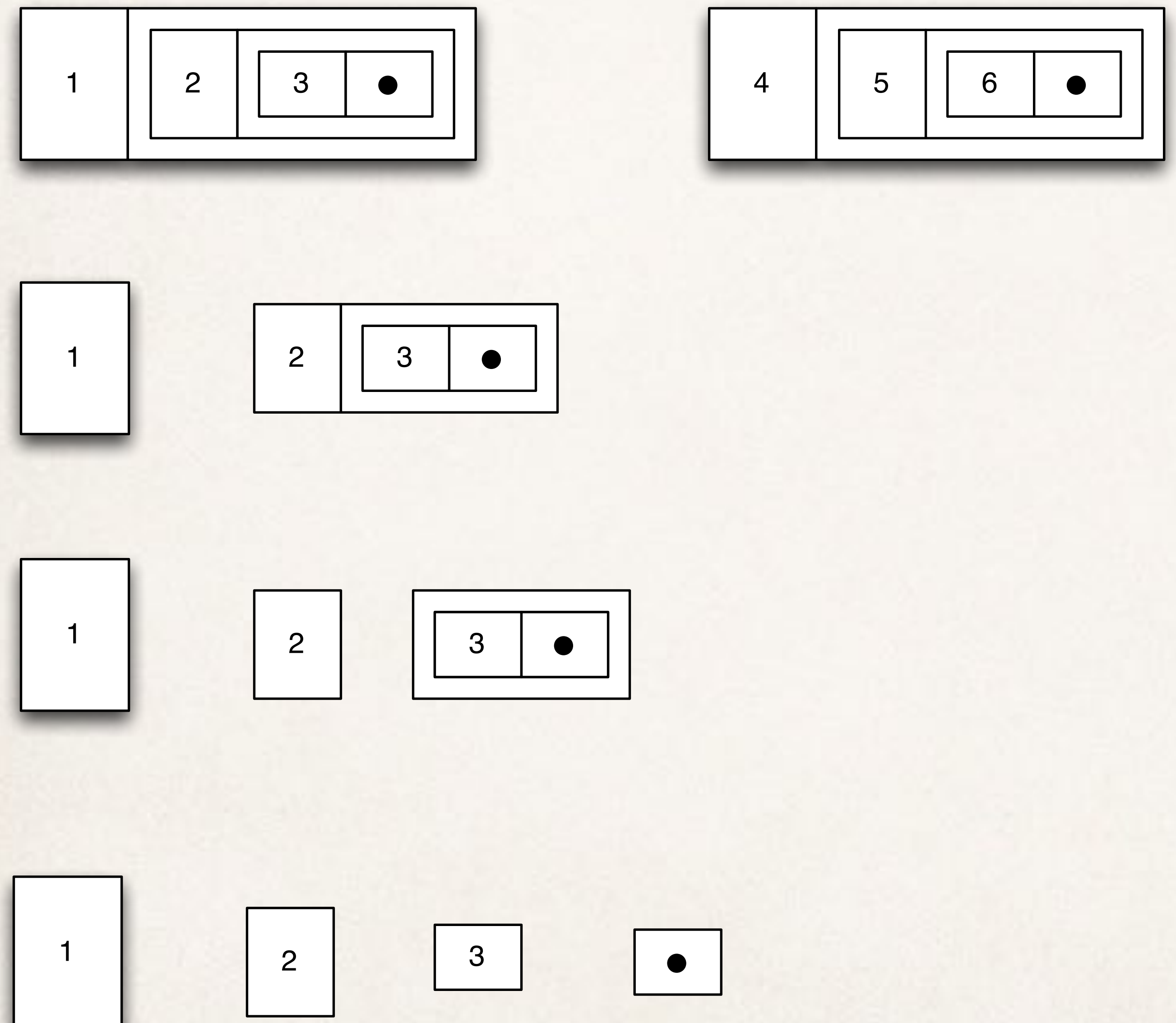
(append (list 1 2 3) (list 4 5 6))



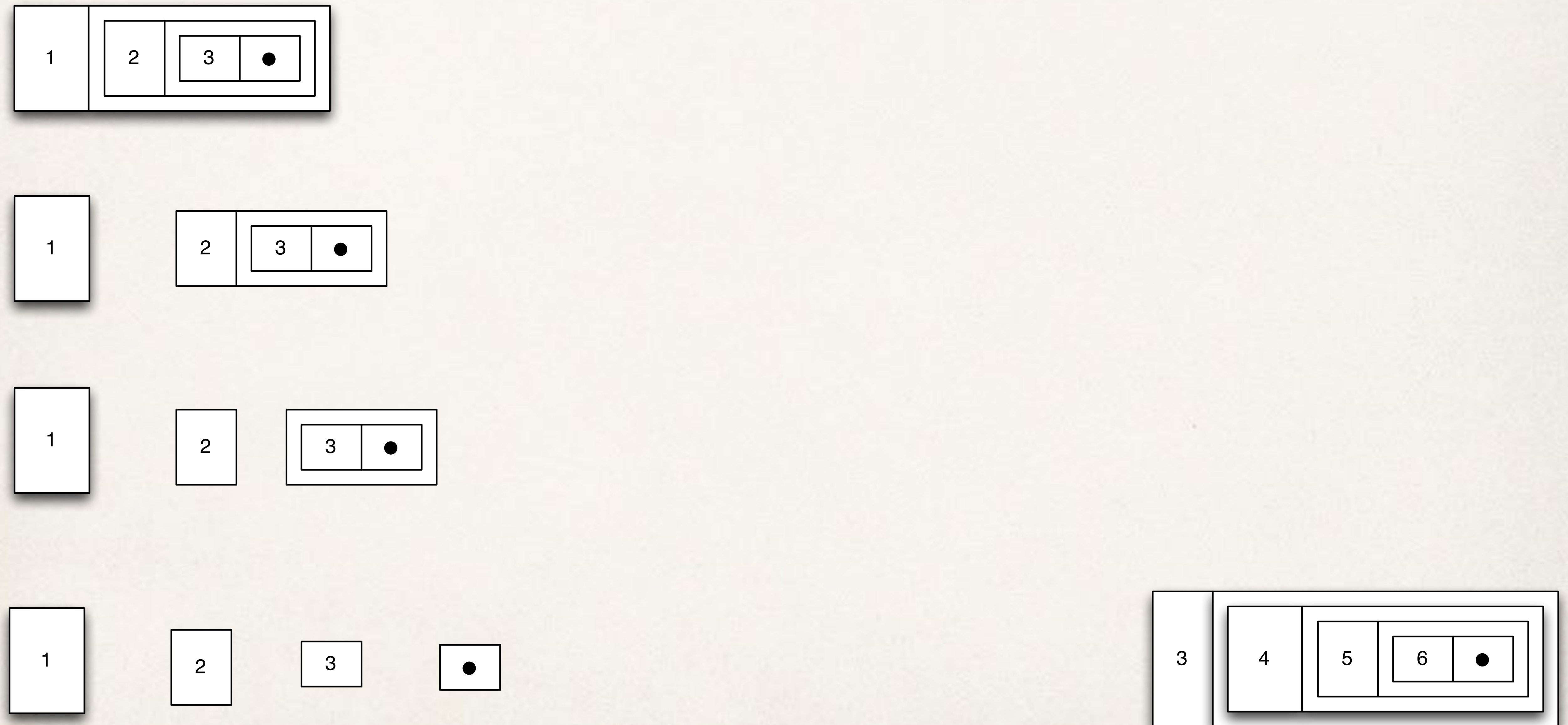
(append (list 1 2 3) (list 4 5 6))



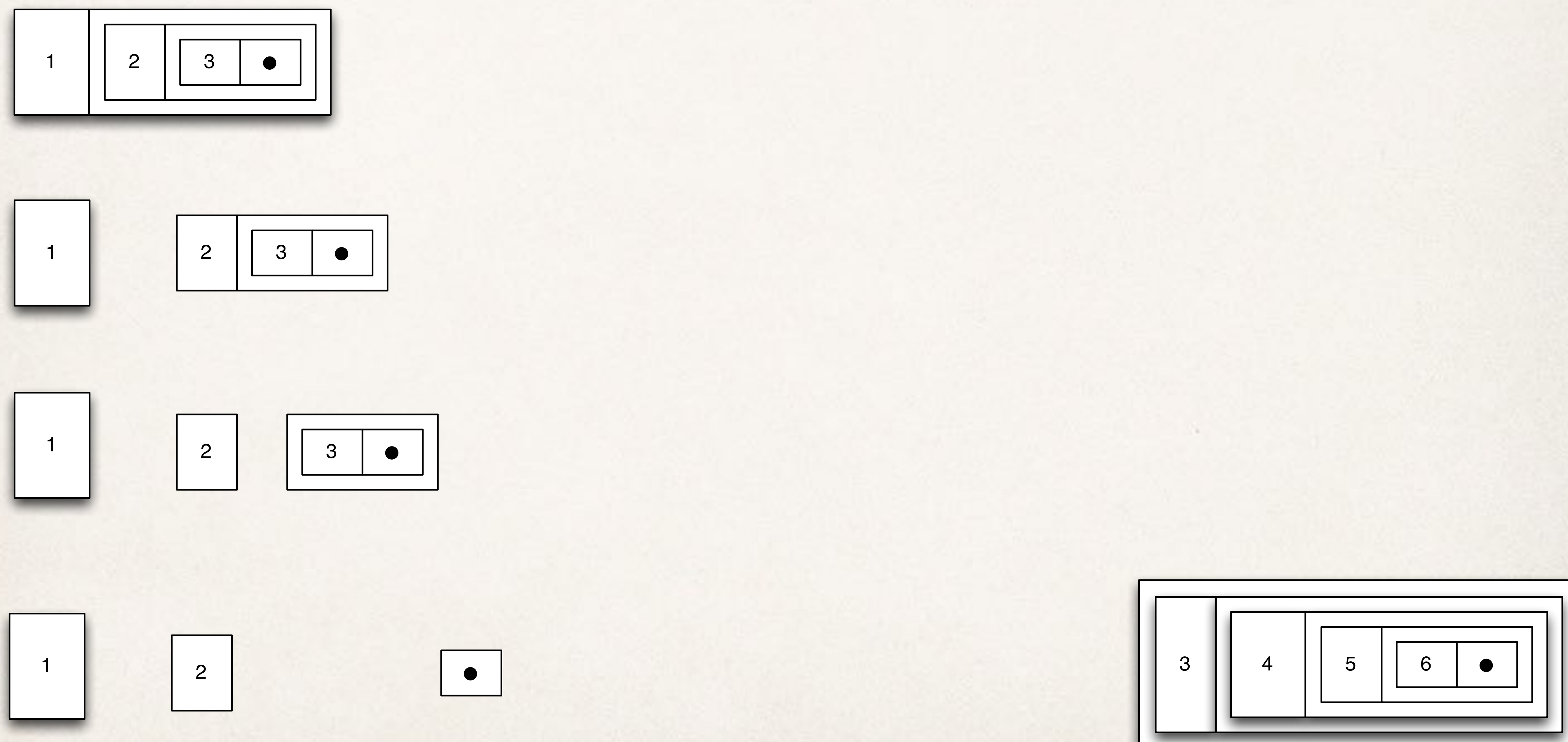
(append (list 1 2 3) (list 4 5 6))



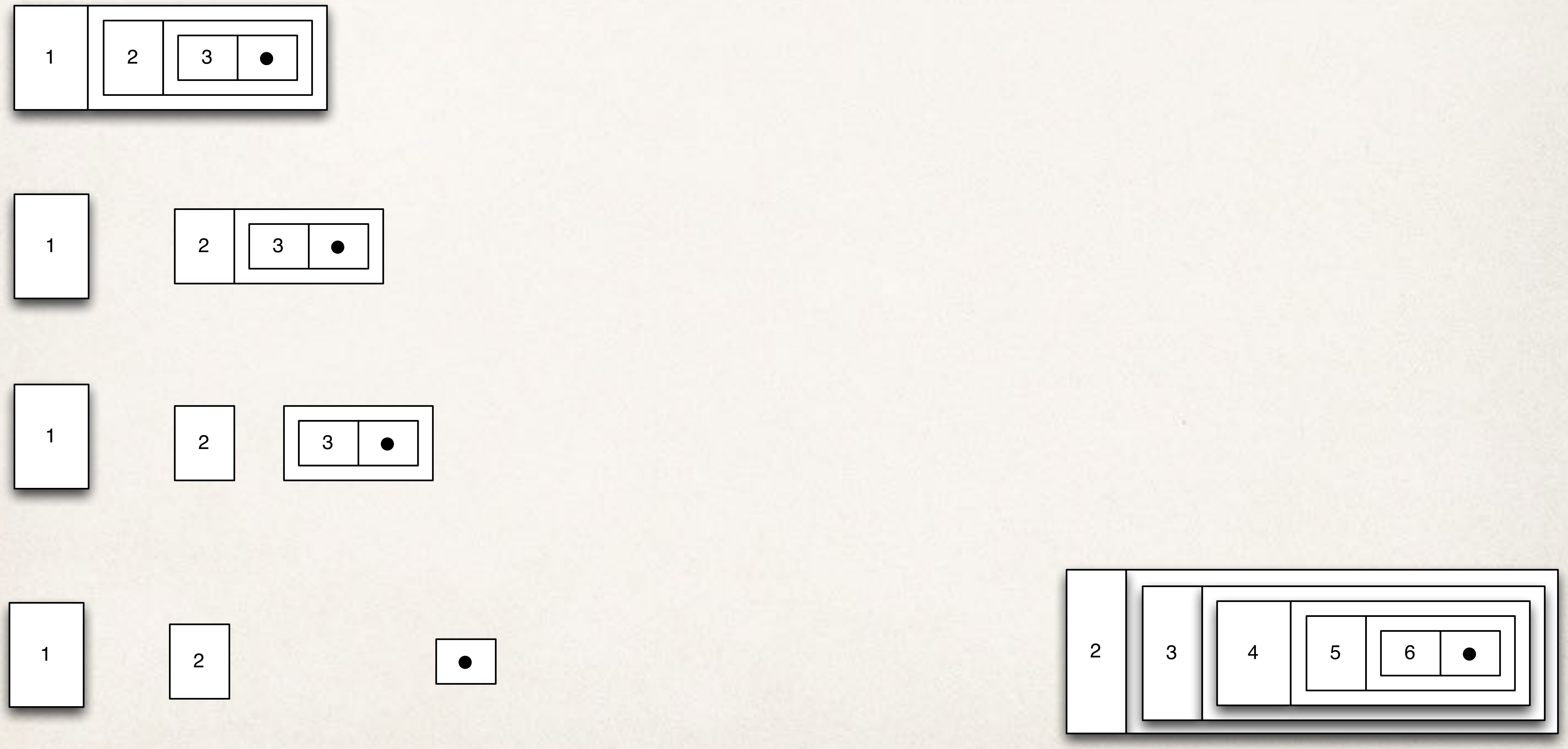
(append (list 1 2 3) (list 4 5 6))



(append (list 1 2 3) (list 4 5 6))

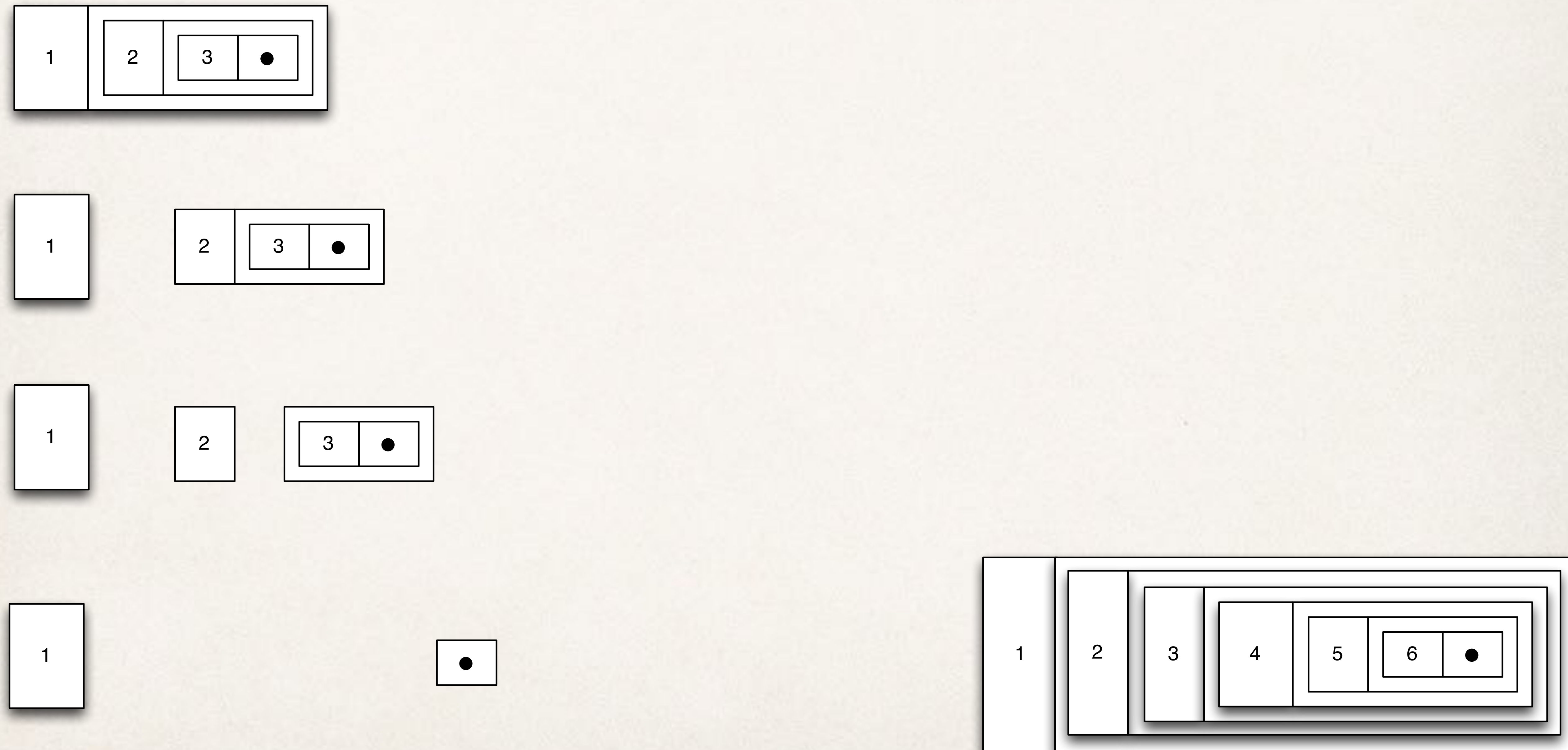


(append (list 1 2 3) (list 4 5 6))



(append (list 1 2 3) (list 4 5 6))

(append (list 1 2 3) (list 4 5 6))



Creating lists…of squares

- The perfect squares:

```
(define (square-list k)
  (if (= k 0)
      (list 0)
      (cons (* k k)
            (square-list (- k 1)))))
```

Note: **element** here, but **list** here

- Hmm…it lists them backwards!

```
> (square-list 4)
(16 9 4 1 0)
```

A list user's guide…

- Suppose that `L` is a list in Scheme;
  - then you can tell if it is empty by testing `(null? L)`; if not…
  - its first element is `(car L)`;
  - the "rest" of the elements are `(cdr L)` (this is a list, and might be empty).

- Suppose that `L` is a list in Scheme and x is a value;
  - `'()` or `(list)` is the empty list.
  - `(cons x L)` is a new list—its first element is x; the rest of the elements are those of L.
  - The list containing only the value x? Same idea, but use the empty list for L: `(cons x '())` or `(list x)`.

Squares in the right order

- It's easy if both ends of a range are given: (why did this make it easy?)

```
(define (squares start finish)
  (define (square x) (* x x))
  (if (> start finish) ‘()
      (cons (square start)
            (squares (+ start 1) finish))))
```

- We can wrap this in a definition that starts at zero:

```
(define (forward-squares k)
  (define (square x) (* x x))
  (define (squares start finish)
    (if (> start finish) '()
        (cons (square start) (squares (+ start 1) finish))))
  (squares 0 k))
```

Mapping a function over a list

- Applying function to each element of a list is called *mapping*. It's a powerful tool.

```
(define (map f items)
  (if (null? items)
      '()
      (cons (f (car items))
            (map f (cdr items)))))
```

- Then, for example:

```
> (map (lambda (x) (* x x)) '(0 1 2 3 4 5 6))
'(0 1 4 9 16 25 36)
> (map (lambda (x) (* x x)) '())
'()
```

Back to asymmetry: Reversing a list. Not as easy as you thought...

- Reversing a list. One strategy: peel off the first element; reverse the rest; append the first element to the end. This yields:

```
(define (reverse items)
  (define (append list1 list2)
    (if (null? list1)
        list2
        (cons (car list1)
              (append (cdr list1) list2))))
  (if (null? items)
      '()
      (append (reverse (cdr items))
              (list (car items)))))
```

Then...

```
> (reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

Even after all that work: This `reverse` has a serious problem

- How long does it take to reverse a list? (One good way to measure the running time of a SCHEME function is to measure the total number of procedure calls it generates.)
- If the list has n elements, `reverse` is called on each prefix. There are about n of these, which looks OK.
- However, each `reverse` also calls `append`. If `reverse` is called with a list of k elements, the append needs to step all the way through this list in order to get to the end, generating k total calls to `append`.
- All in all, this is roughly $n + (n-1) + \ldots + 1$ calls; about $n^2/2$. Surely we can reverse a list in roughly n steps!

Appending the car once the rest of the list is reversed is costly...

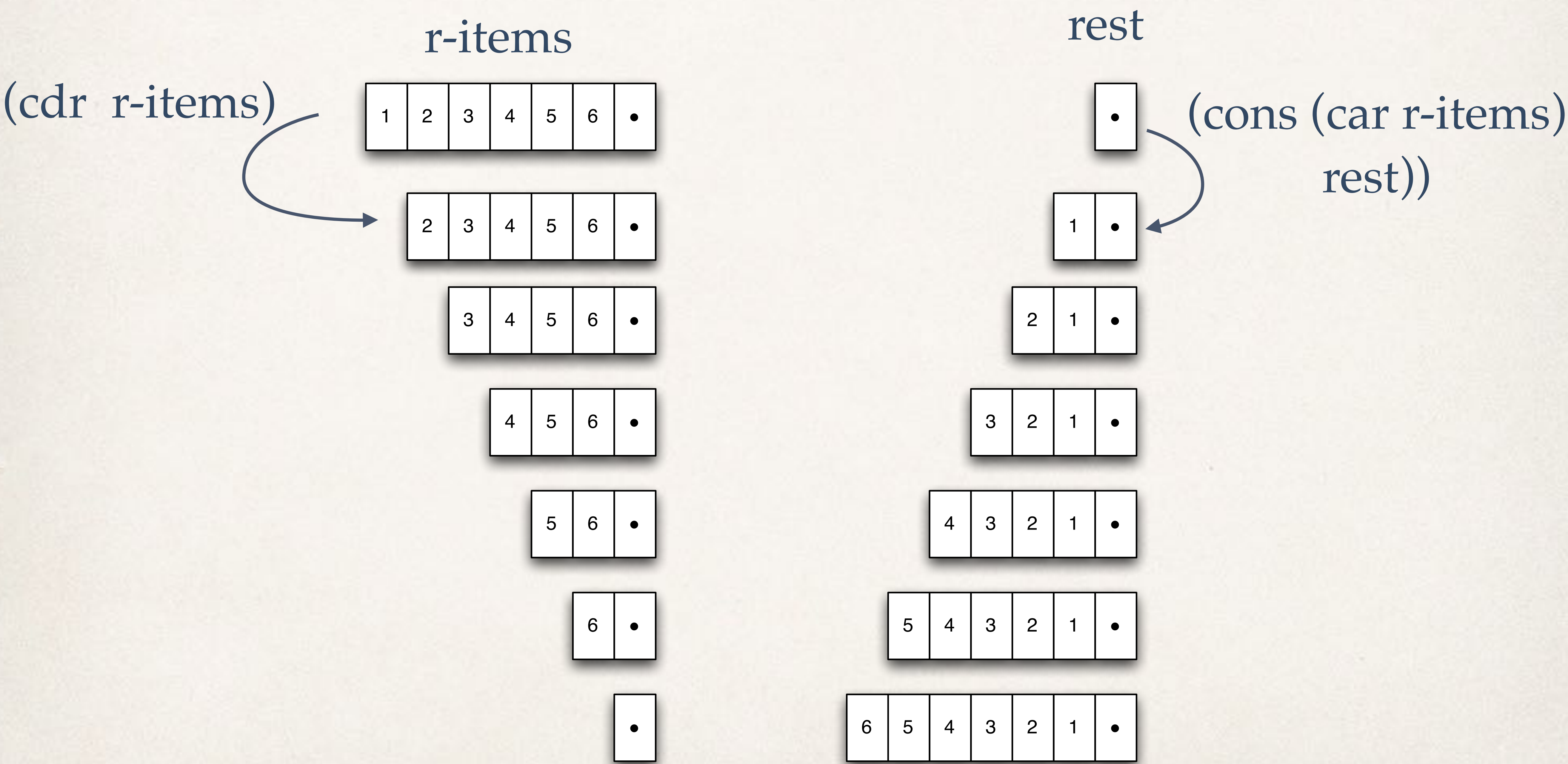
- ...what if we pass the car along as a parameter, asking our next-in-line to take care of the job of appending it to the resulting list?

- Specifically, consider the function `(reverse-and-append list rest)`: it should reverse `list`, append `rest` onto the end, and return the result.

```
(define (reverse-and-append r-items rest)
  (if (null? r-items)
      rest
      (reverse-and-append (cdr r-items)
                          (cons (car r-items) rest))))
```

- Note: this simply generates ~n recursive calls!

Visually



r-items
rest
(cdr r-items)
(cons (car r-items) rest))
1 2 3 4 5 6 •
•
2 3 4 5 6 •
1 •
3 4 5 6 •
2 1 •
4 5 6 •
3 2 1 •
5 6 •
4 3 2 1 •
6 •
5 4 3 2 1 •
•
6 5 4 3 2 1 •

Sorting a list of numbers: Insertion Sort

- Goal
  - Sort a list of value in increasing order
- Idea
  - Find the minimum,
  - Extract it (remove it from the list),
  - Sort the remaining elements,
  - Add the minimum back in front!

Finding the smallest

- Objective
  - Write a function that finds the smallest element in a list
- Inductive definition
  - Base case?
    - List of one element....
  - Induction?
    - Smallest between the head and smallest in tail

The Scheme code

- One auxiliary function to choose the smallest element
- One plain induction on the list.

```
(define (smallest l)
  (define (smaller a b) (if (< a b) a b))
  (if (null? (cdr l))
      (car l)
      (smaller (car l) (smallest (cdr l)))))
```

Removing from a list

- Goal
  - Remove a *single occurrence* of a value from a list
- Inductive definition
  - Base case:
    - Easy: empty list
  - Induction:
    - If we have a match: done! Just return the tail.
    - If we don't: remove from the tail and preserve the head.

The Scheme code

- One plain induction on the list.
  - *v*: the value to remove
  - *elements*: the list to remove it from

```
(define (remove v elements)
  (if (null? elements)
      elements
      (if (equal? v (car elements))
          (cdr elements)
          (cons (car elements)
                (remove v (cdr elements))))))
```

Putting the pieces together to sort

- Use smallest and remove!

```
(define (selSort l)
  (if (null? l)
      '()
      (let* ((first (smallest l))
             (rest (remove first l)))
        (cons first (selSort rest)))))
```

- Use a `let*`
  - To first bind `first` to the smallest element of the list;
  - Then *use* `first`'s value to trim the list.

And…to minimize clutter

```
(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 rest)))))
```

No need to traverse the list twice; one pass extraction & minimization

- Goal
  - *Find* and *Extract* the smallest element from a list (in one pass!).
- Idea
  - Return *two* things (a pair!)
    - The extracted element
    - The list without the extracted element.

Minimization and Extraction in One Sweep

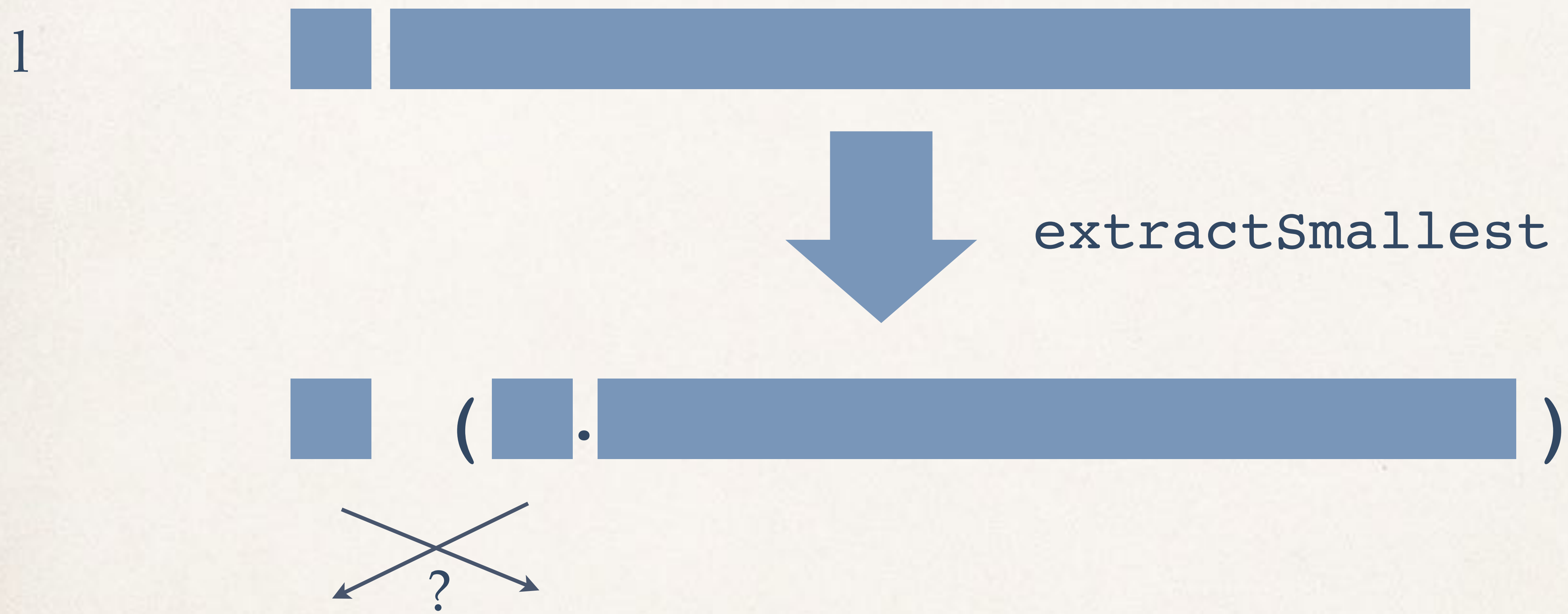
- To improve readability, we introduce *convenience* functions to make & consult pairs.
- Reserve `cons`/`car`/`cdr` for list operations

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

(define (extractSmallest l)
  (if (null? (cdr l))
      (make-pair (car l) '())
      (let  ((p (extractSmallest (cdr l))))
        (if (< (car l) (first p))
            (make-pair (car l)    (cons (first p) (second p)))
            (make-pair (first p) (cons (car l)    (second p)))
            ))))
```

Assume l has at least one element

The Picture



Reassemble, depending on which is smaller...

Then Selection Sort is easy...

- Use the combined find and extract

```
(define (selSort l)
  (if (null? l)
      '()
      (let ((p (extractSmallest l)))
        (cons (first p) (selSort (second p))))))
```

- extractSmallest returns a pair
  - Pick the first as the value to place in front
  - Pick the second as the trimmed list to recur on.

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.

Elements needing to be handled

Already sorted elements

```
(define (sort unexamined sorted)
  ...)
```

A Solution using Accumulators

Smallest so far

Unexamined elts

Examined elts

```
(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) '()))
```

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.

What about another *ordering*?

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

Idea

- Externalize the ordering!
- 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)
        (smaller (car l) (smallest (cdr l)))))
  (define (remove v l)
    (if (null? l)
        l
        (if (equal? v (car l))
            (cdr l)
            (cons (car l) (r
  (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?

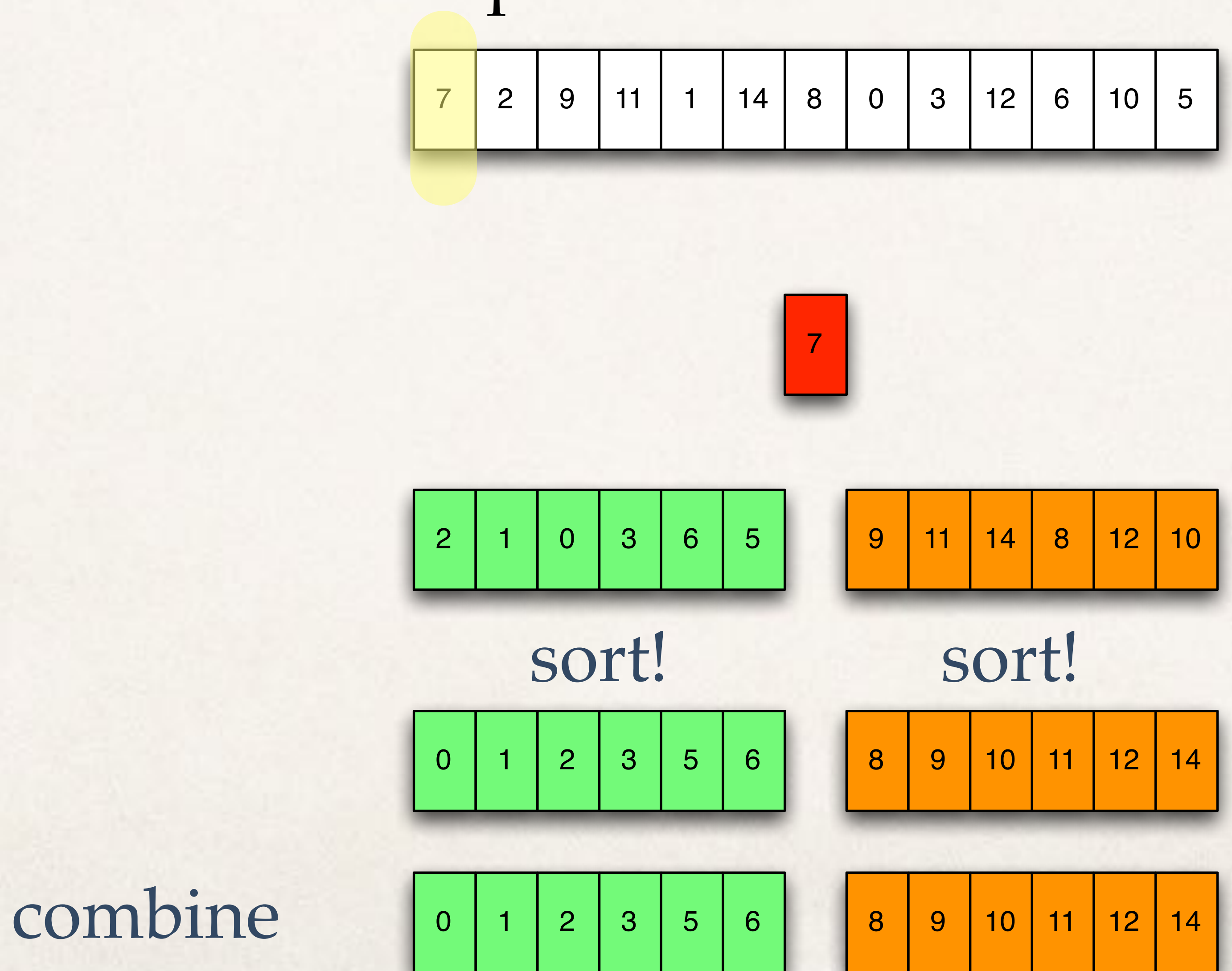
Closure

- 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?**

Let's *Quick*Sort

- Algorithm design idea
  - Divide and Conquer!



Key Ingredients

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

Partitioning

Why are we using accumulators for left/right?

- Recursive definition
  - Base case: empty list
  - Induction: Deal with one element from the list
  - Returns: a pair (the two partitions)

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

*Quick*Sort

- Also Recursive
  - Base case: empty list
  - Induction: partition & sort

Why are we using let* ?

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

Cleanup

- Once again, you can *hide* partition inside quickSort
  - After all, it is used only by quickSort....
- Once again, you can *externalize* the ordering
  - Use a function for comparisons.
  - Pass it down to quickSort!