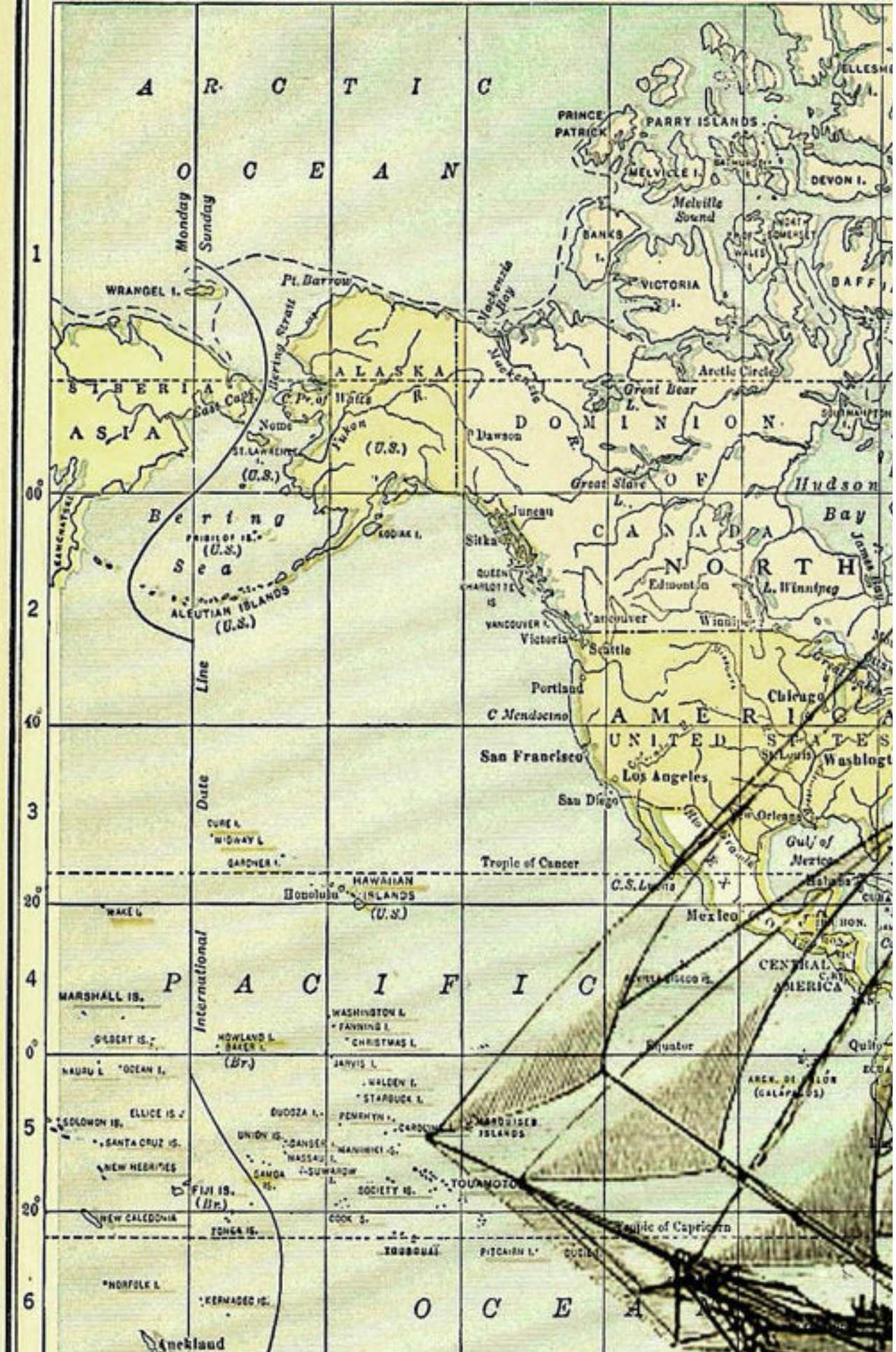
HEIRARCHICAL DATA

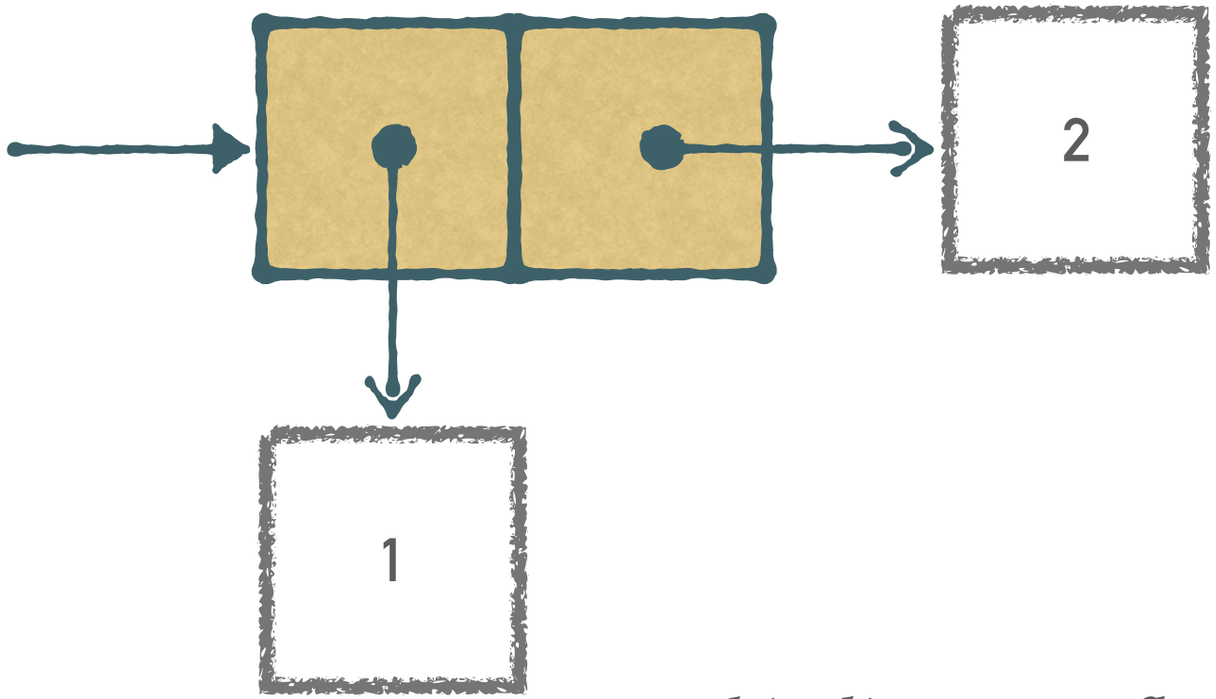
Closure property



USING PAIRS

- As we've seen, it's possible to use a pair or multiple pairs to represent many different types of data
- We can diagram pairs and their layout using *box-and-pointer* diagramming

A SIMPLE BOX-AND-POINTER DIAGRAM



This diagram reflects a structure created with

(cons 1 2)

The directionality of the arrows has ***no*** *meaning*

The arrow coming in from the left indicates the starting point of the diagram

A SLIGHTLY MORE COMPLEX DIAGRAM

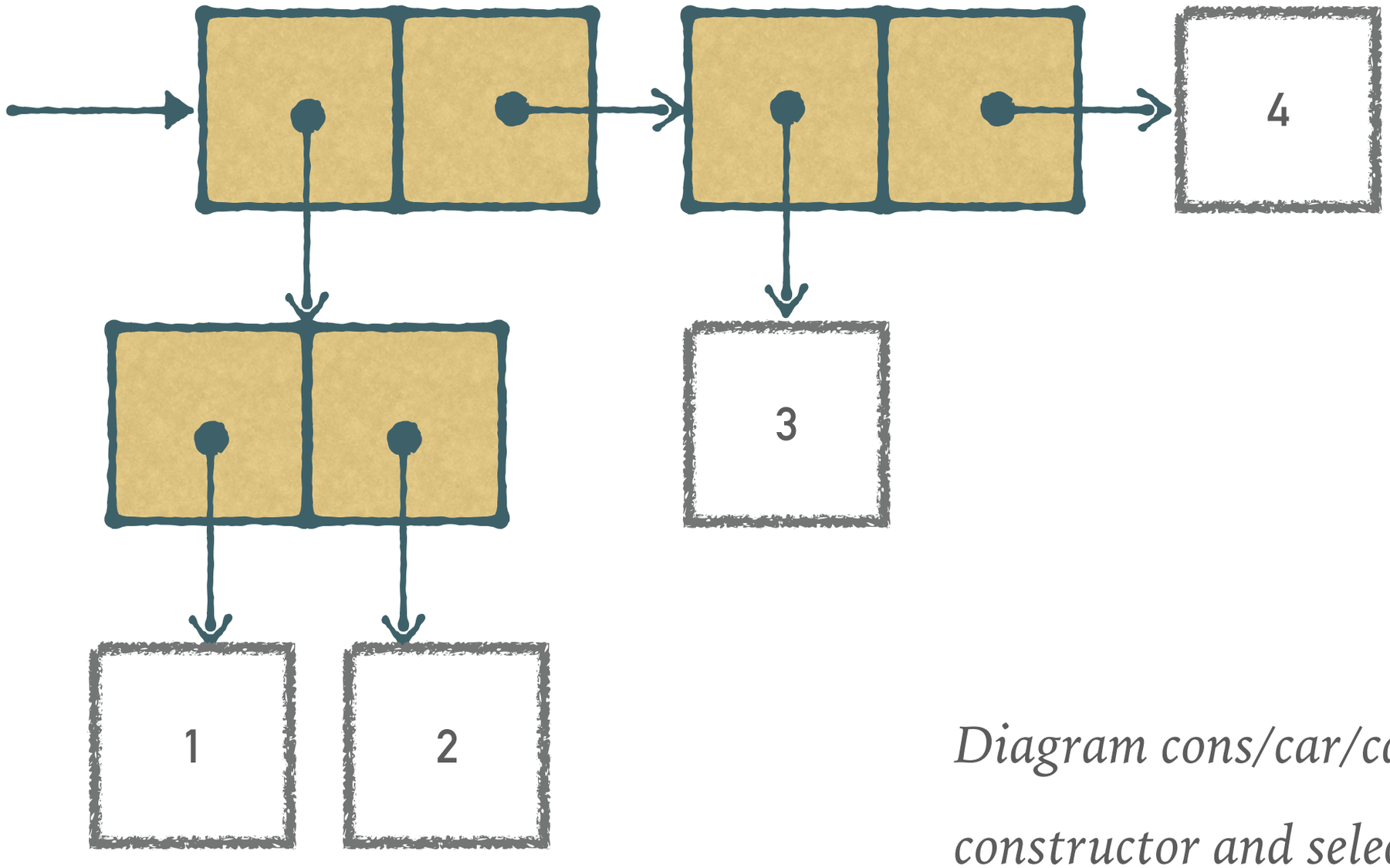


Diagram cons/car/cdr for the constructor and selectors for this data structure

A SLIGHTLY MORE COMPLEX DIAGRAM – VARIANT

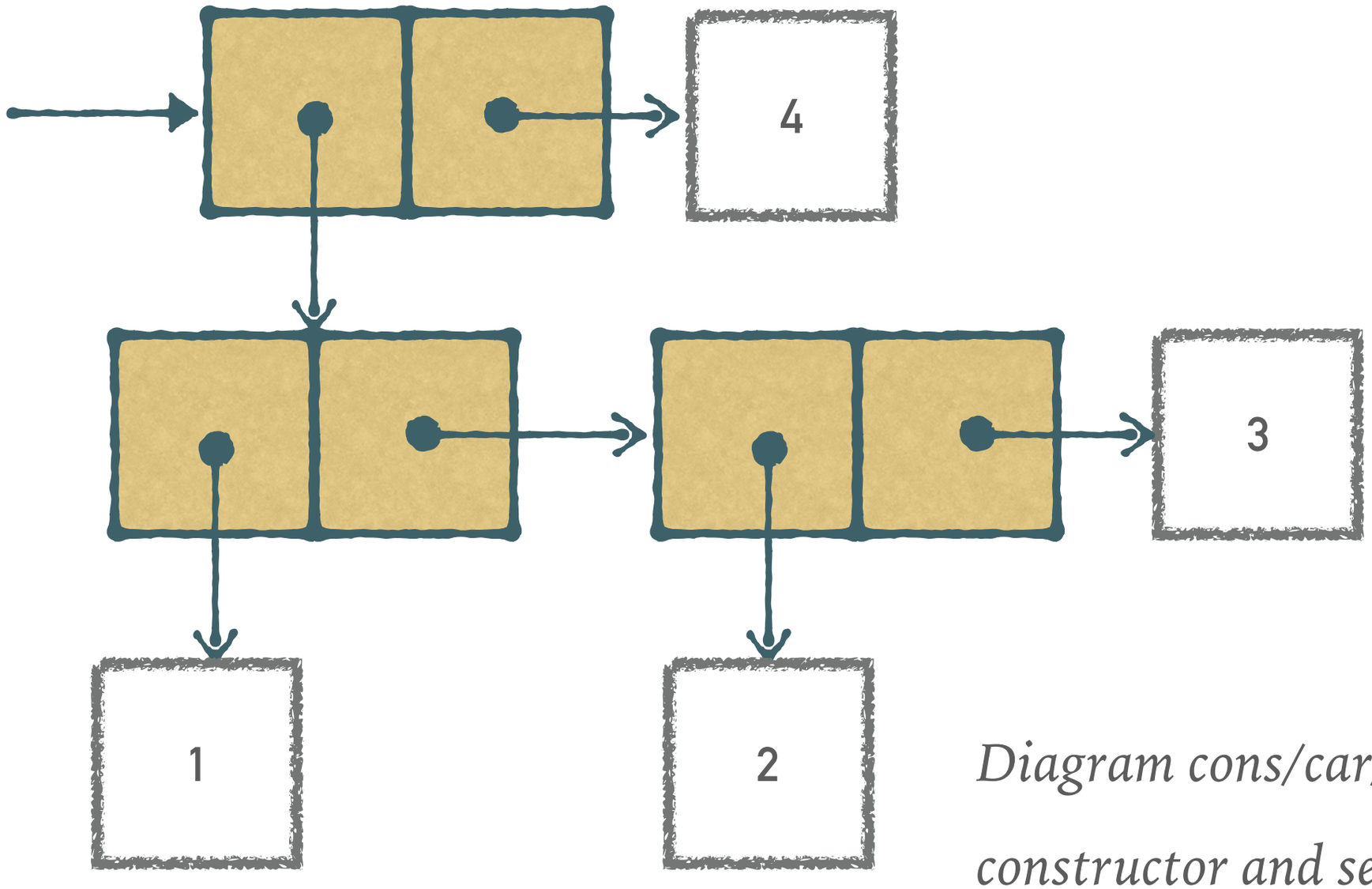
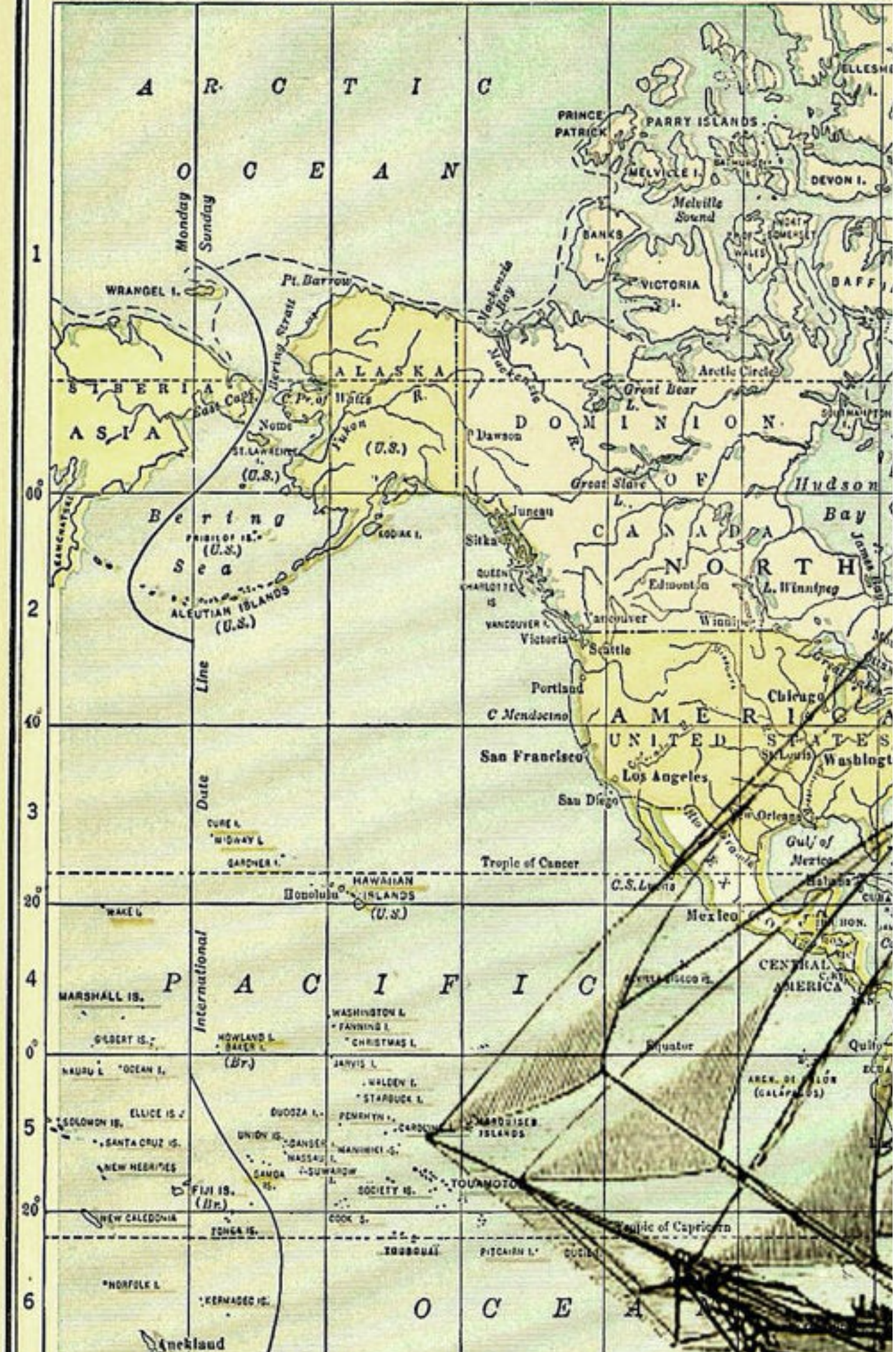


Diagram cons/car/cdr for the constructor and selectors for this data structure

WHICH DIAGRAM IS MORE CORRECT?

- That's a trick question. In and of itself, the *pair* structure does not have enough information to make one implementation better or worse than the other.
- Either structure could be the "correct" one.
- Each can be correct given a specific use case!

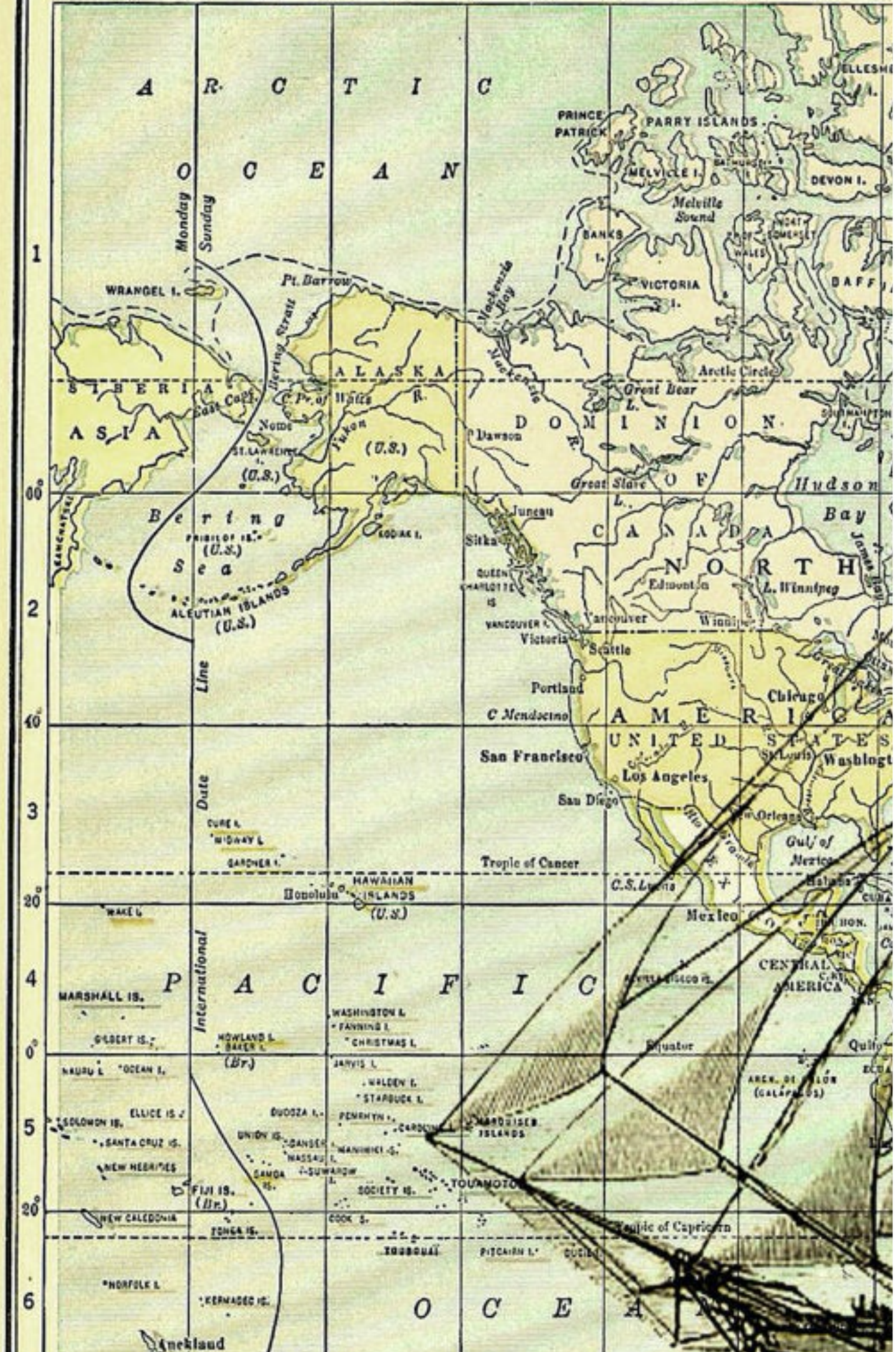


CLOSURE

- Most importantly, we're able to use a pair to create pairs
- Using the same mechanism to compose items which are themselves composed with a particular mechanism is known as the *closure property*.
- Closure allows us to create complex hierarchical structures

SEQUENCES

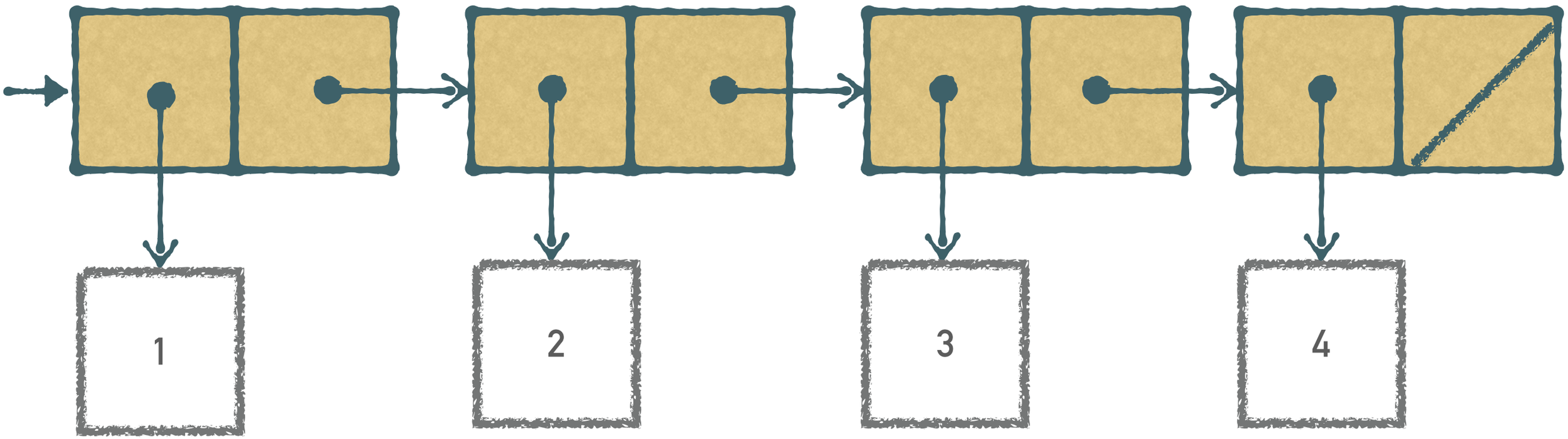
East Greenland Sea
GREENLAND
(Danish)
Barents Sea
BAFFIN BAY
BAFFIN ISLAND
DEVON I.
Hudson Bay
Davis Strait
Denmark Strait
ICELAND
BRITISH ISLES
Arkhangelsk
UNION
EUROPE
ALGERIA
LIBYA
EGYPT
WEST INDIES
Caribbean Sea
Gulf of Mexico
CENTRAL AMERICA
ATLANTIC
OCEAN
SOUTH AMERICA
BRAZIL
BELGIAN CONGO
ANGOLA
Washington
New Orleans
Chicago
Rio de Janeiro
Drake Strait
COUNTRIES AND THE
United States
British
French
Belgian
Dutch



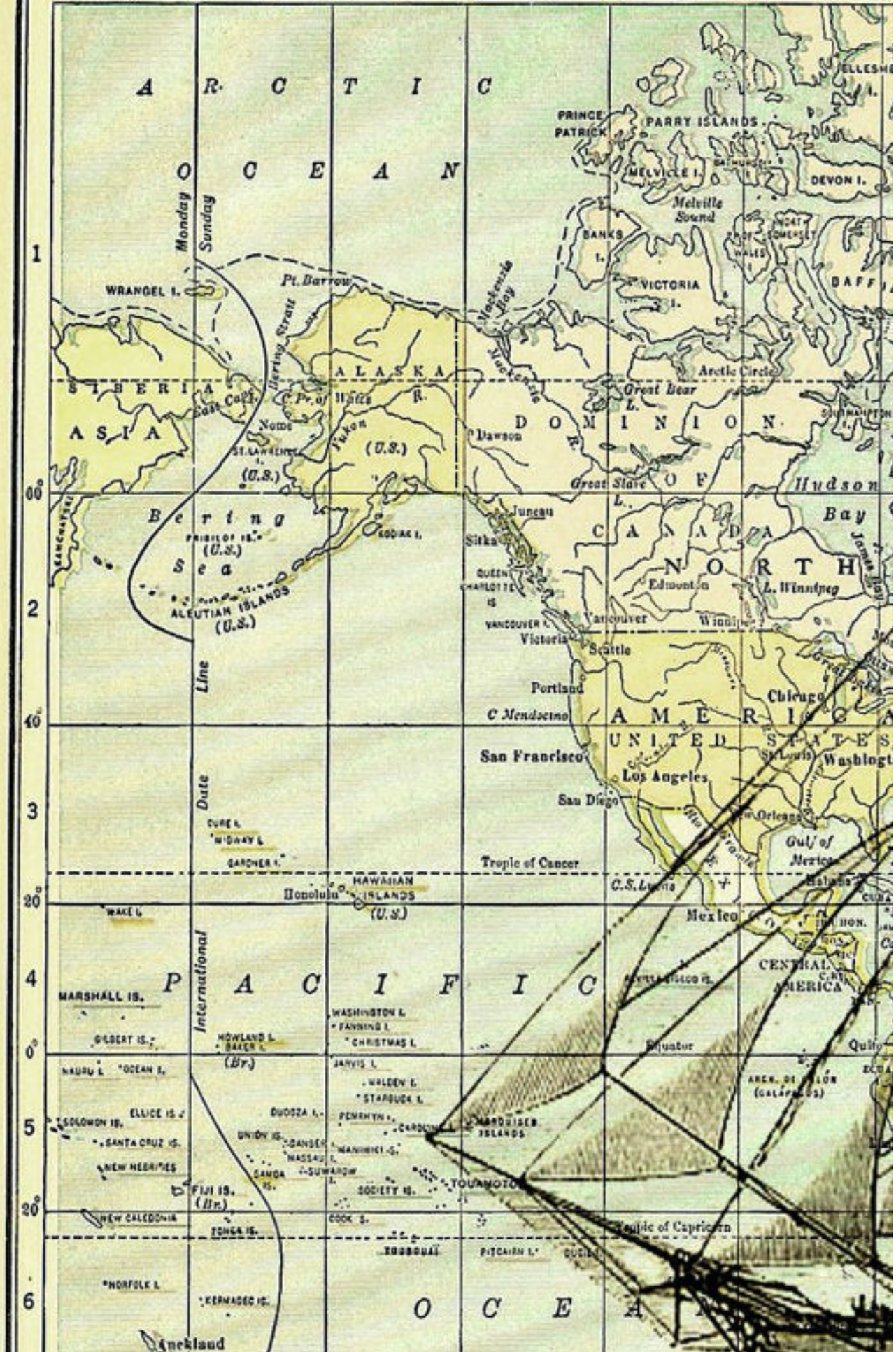
SOME STRUCTURES ARE MORE EQUAL THAN OTHERS

- While it's possible to build a representation of data using pairs and create it in a custom way each time, some structures have similarities each time we build them
- A *list* is one such structure
- A list is based upon a sequence of pairs

LIST



A box with a slash through it is to be read as 'nil'



NIL

- Before we talk about lists in general, let's discuss the special value of *nil*
- *Nil* is short for *nihil*, which means *nothing*
- We use *nil* to indicate the end of a list
- Can you think of other programming languages that use *nil* to indicate the end of a list?

CREATING A LIST

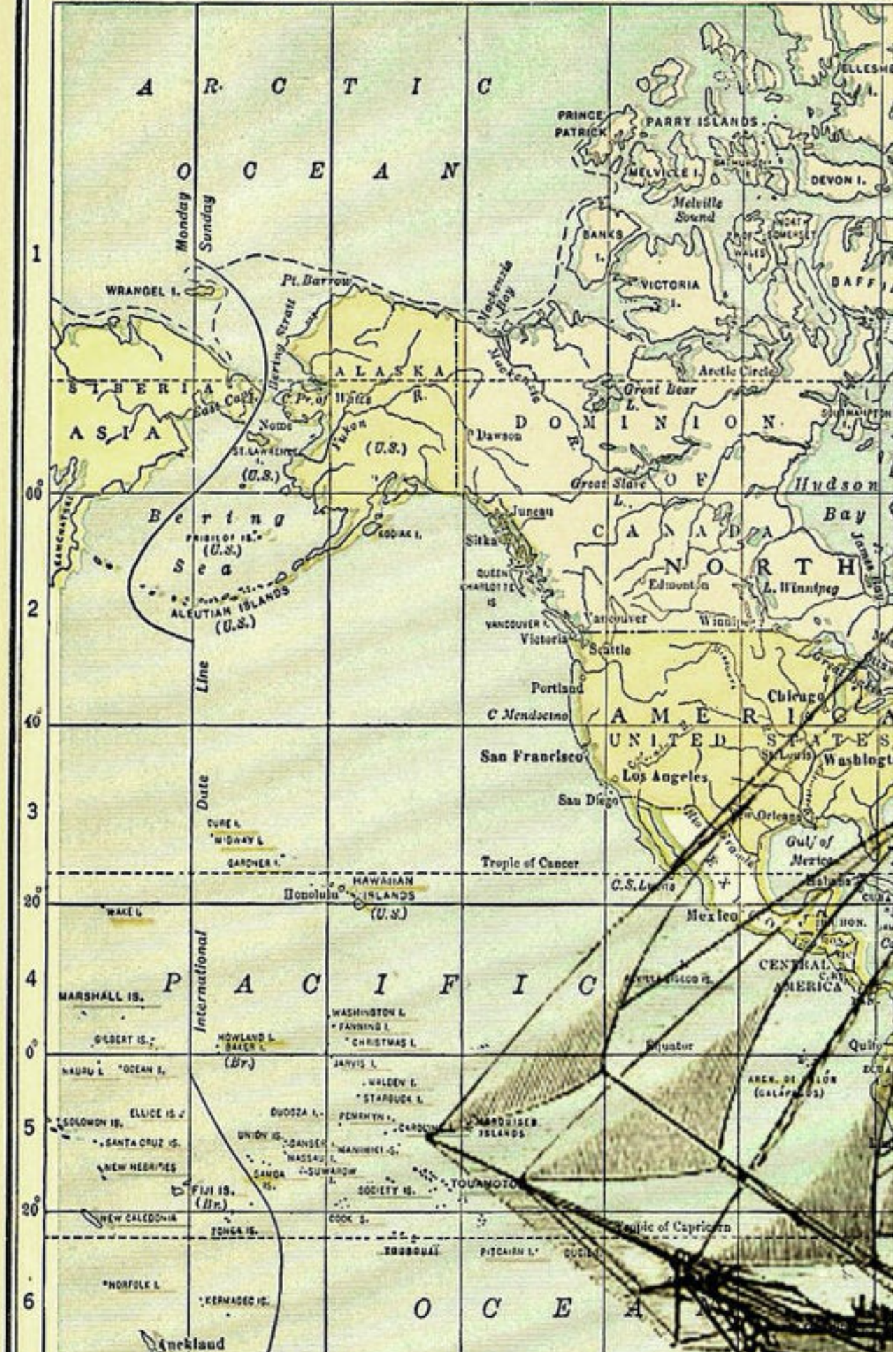
```
(define one-through-four (list 1 2 3 4))


# is exactly the same as…
```

```
(define one-through-four
              (cons 1
                    (cons 2
                          (cons 3
                                (cons 4 nil)))))
```

EXTRACTING VALUES

- To get values from a sequence, we use *car* and *cdr*, just like in a pair
- *car* will provide the first item in a list
- *cdr* will provide the remaining items in the list



TESTING LISTS

- As I know that many of you may wish to test your implementations of various homework problems (or even just see if my source is correct), there's an addendum to your textbook SICP:
- *nil* is no longer natively defined in Scheme
- You can easily provide your own implementation of *nil* by providing an empty list

YOUR OWN NIL

```
(define nil ())
(define x (cons 1 (cons 2 (cons 3 nil))))


# which of course is the same as

(define x (cons 1 (cons 2 (cons 3 ()))))


# why might the first implementation be
preferable to the second? why not?
```

EXTRACTING VALUES

```
(define y (cons 1 (cons 2 (cons 3 nil))))


(car y)
```

1

```
(cdr y)
```

(2 3)

```
(car (cdr y))
```

2

DON'T CONFUSE LISTS WITH THE VALUE OF THE LIST

```
(list 1 2 3 4) is not the same as (1 2 3 4)


# What does (1 2 3 4) mean?
```

MORE SAMPLES

```
(define a (list 1 2 3 4))


(cons 20 a)
```

MORE SAMPLES

```
(define a (list 1 2 3 4))


(cons 20 a)
```

(20 1 2 3 4)

MORE SAMPLES

```
(define a (list 1 2 3 4))


(cons 20 a)
```

(20 1 2 3 4)

```
(cons 10 (car a))
```

MORE SAMPLES

```
(define a (list 1 2 3 4))


(cons 20 a)
```

(20 1 2 3 4)

```
(cons 10 (car a))
```

(10 1)

OPERATIONS ON A LIST

- Lists can be accepted as formal parameters, just like anything else in Scheme
- Since we have the ability to work through a list as a sequence of pairs, we can work on a sequence by *cdring-down* the list
- Scheme provides an operator *null?* which returns true if a list is empty

EXAMPLE: FIND-ELEMENT

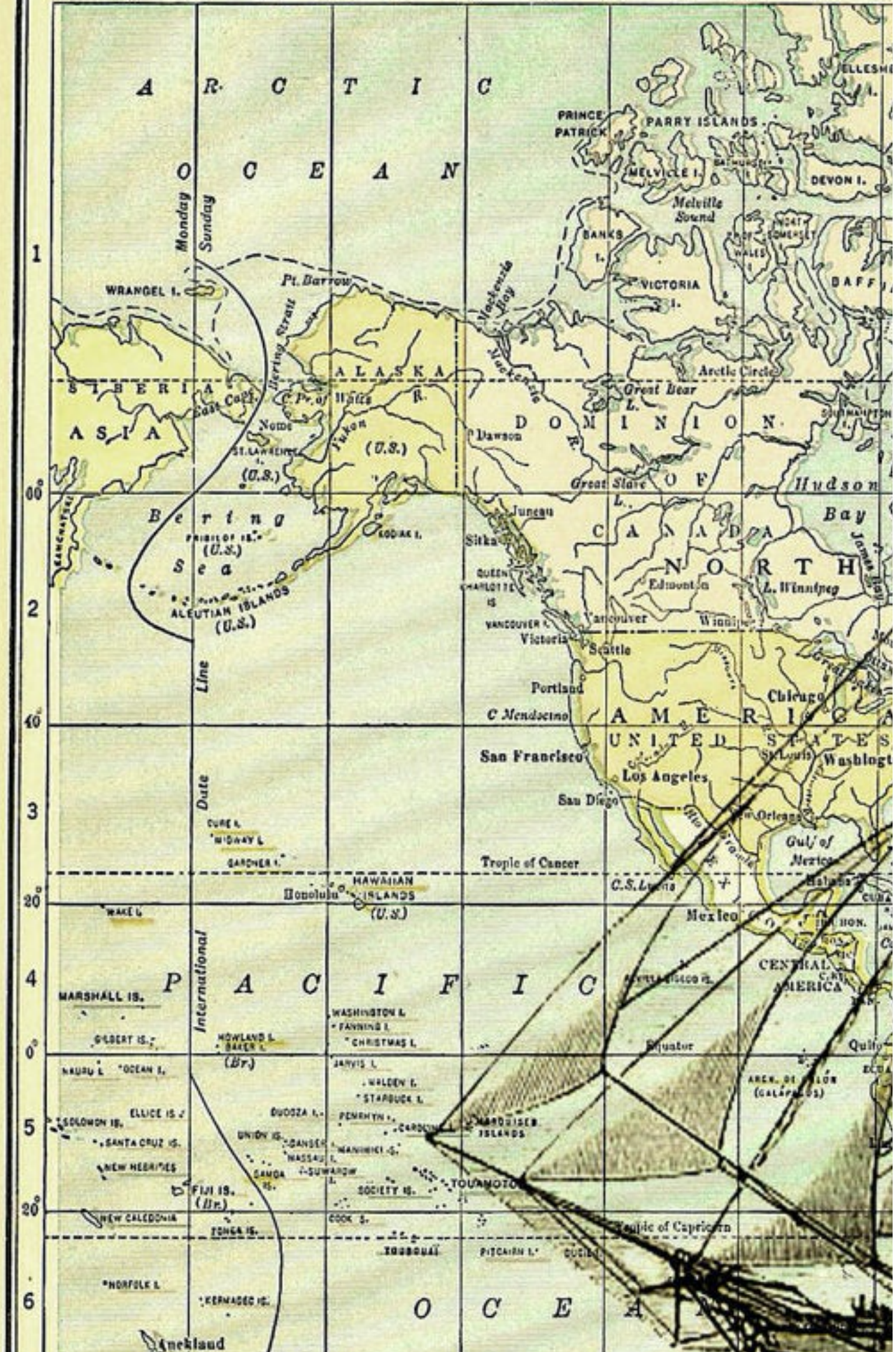
```
(define (find-element items n)
    (if (= n 0)
        (car items)
        (find-element (cdr items) (n-1))))
```

Work through substitution on the board

SAMPLE: LENGTH

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))
```

Perform substitution on the board



MAPPING OVER LISTS

- Not surprisingly, scheme also provides the ability to *map* over a list
- As a reminder, when we *map* a list, we are essentially transforming that list into a list of the same size, but with newly computed values
- *map* is a higher-order function, as we've discussed earlier
- *map* takes a procedure as one of its arguments

SAMPLE MAP

```
(map (lambda (x) (* x x)) (list 1 2 3))
```

DEFINING MAP OURSELVES

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))
```

SAMPLE MAP

```
(map (lambda (x) (* x x)) (list 1 2 3))
```

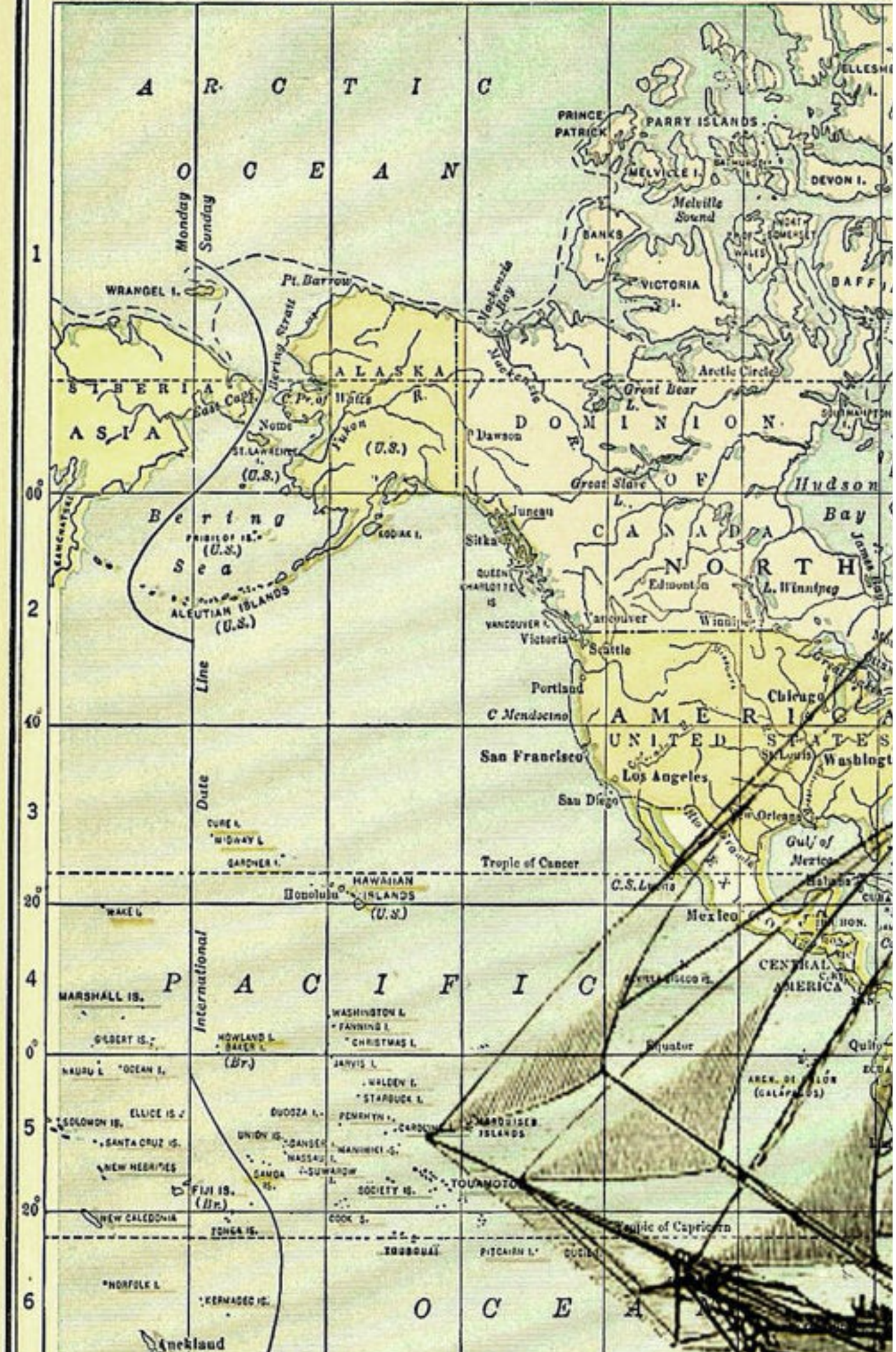
MAP IS INCREDIBLY IMPORTANT

- If for whatever reason you don't feel comfortable with *map* in both Scheme and Rust, please send me an email with questions so that I can clarify it for you
- The most important thing to remember is it returns an *equally sized list* to your original list, with each element transformed by the factor

MID-CLASS REVIEW

- Create a few box-and-pointer diagrams.
  - Make some of them look *silly*, to help reinforce the idea that direction isn't meaningful
- Create a box-and-pointer diagram of a list
- Create a list using *cons*
- Map over a list using a lambda that adds two to the item

HEIRARCHICAL STRUCTURES



SEQUENCE CLOSURES

- Sequences of sequences are absolutely possible due to the *closure property* of a sequence
- What is the closure property?
  - When a structure can be composed of the same type of structure, that is then said to follow the *closure* property
- We can also put a *list* into a pair, of course

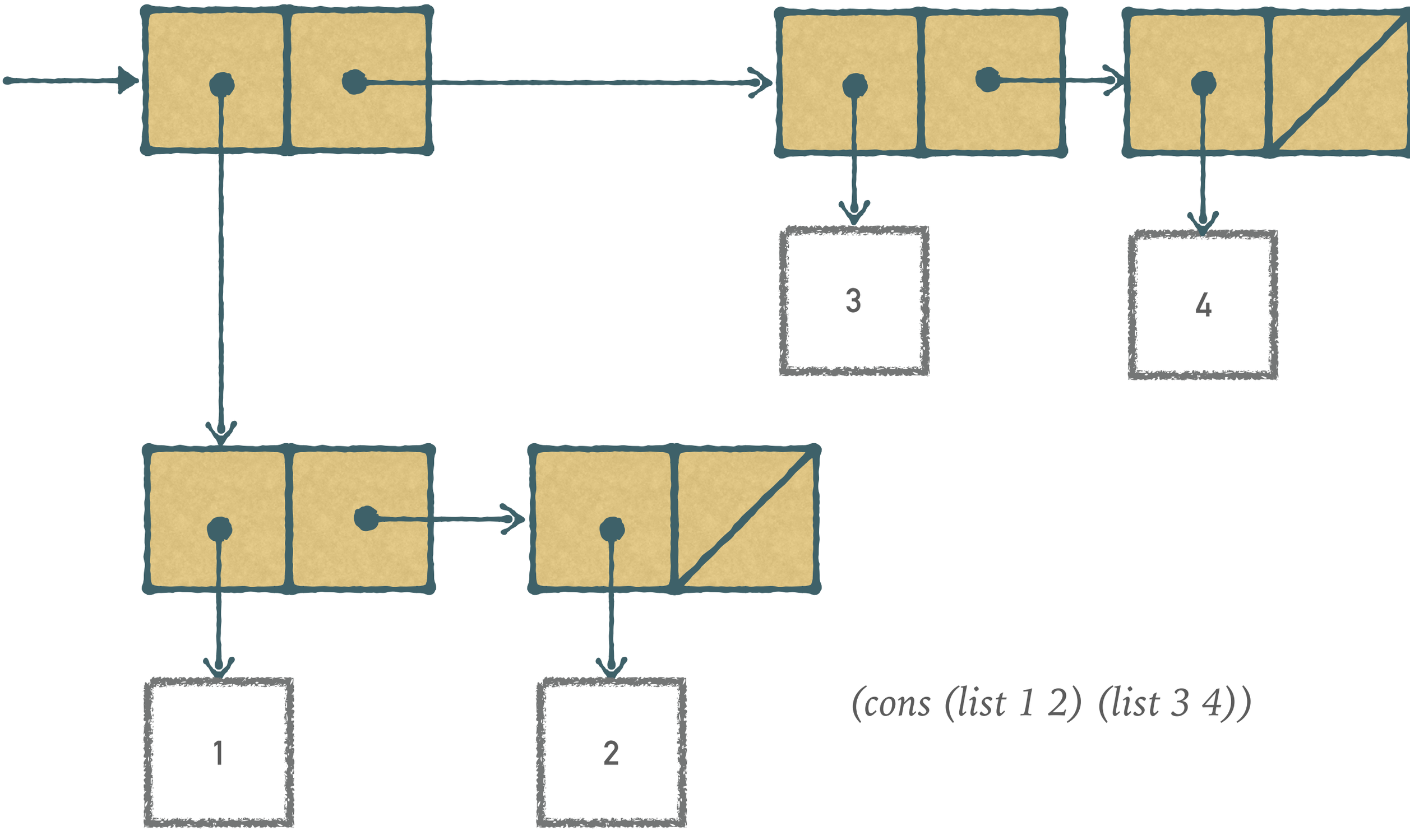
SIMPLE EXAMPLE

```
(cons (list 1 2) (list 3 4))


# becomes
```

((1 2) 3 4)

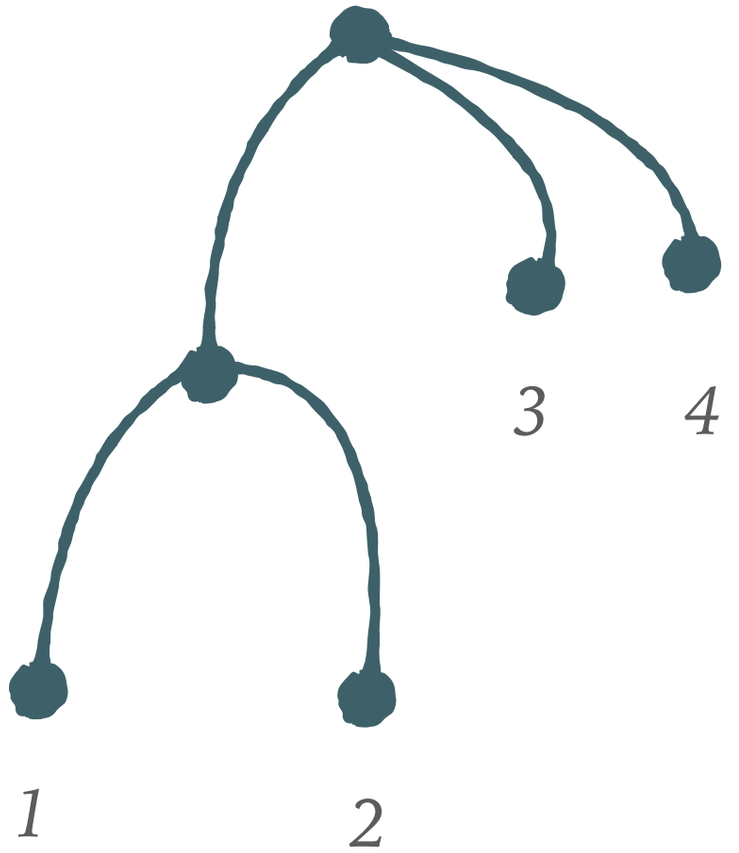
((1 2) 3 4)

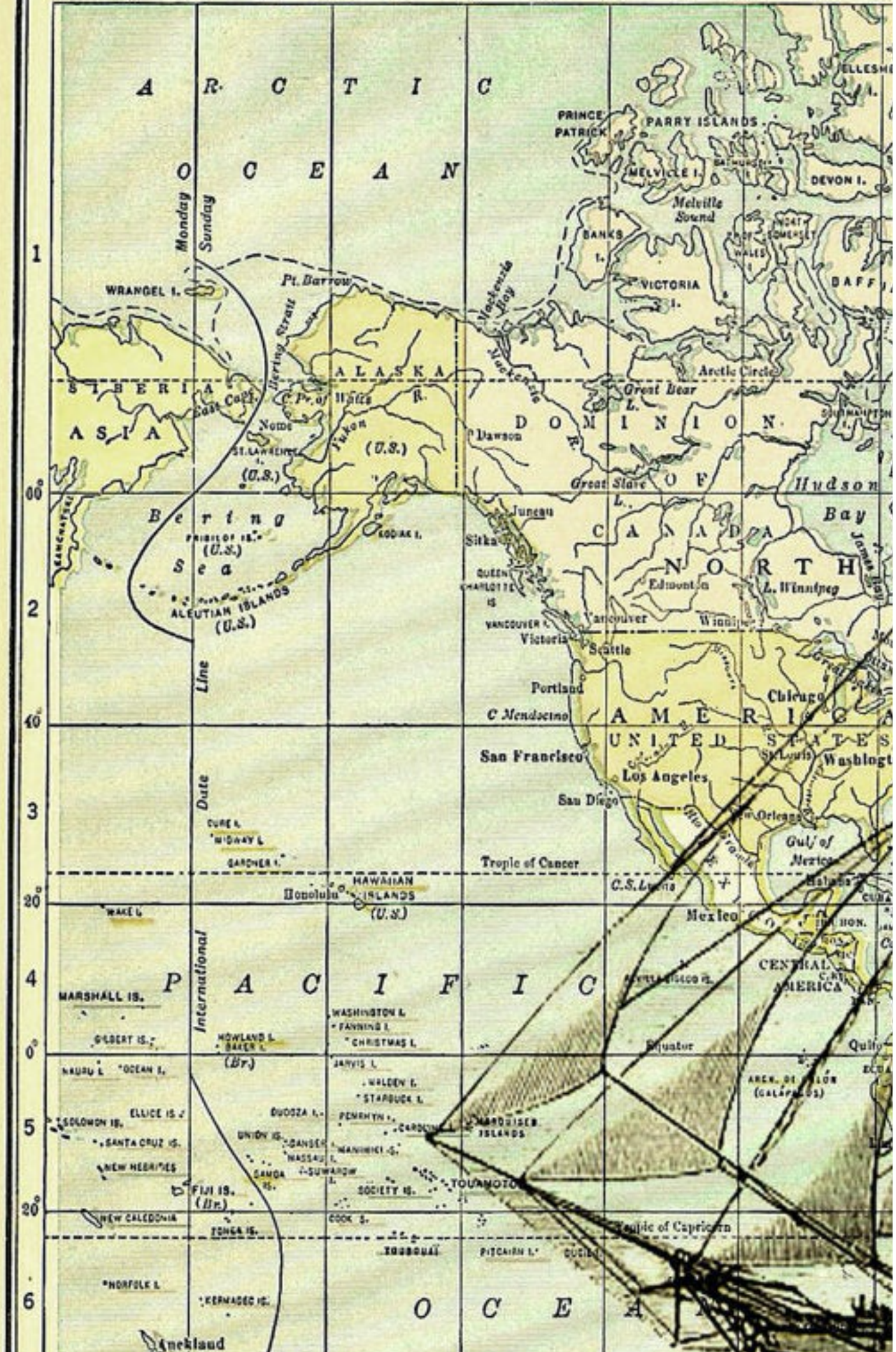


(cons (list 1 2) (list 3 4))

AS A TREE

(cons (list 1 2) (list 3 4))





WORKING WITH TREES

- A natural tool for working with tree structures is recursion, since we can reduce the operation on a tree to operations on its branches, and so on
- We can compute the size of a tree by using a routine like *count-leaves*

COMPUTING THE SIZE OF A TREE

```
(define x (cons (list 1 2) (list 3 4)))
```

```
(length x)
```

```
3
```

```
(list x x)
```

```
(((1 2) 3 4) ((1 2) 3 4))
```

```
(length (list x x))
```

```
2
```

THOSE SIZES DON'T LOOK RIGHT

- Why are these sizes not what we would expect?
- The first item in the overall list is itself a list!
- We need to traverse the tree to properly compute its size
- If the *length* of a list is 1 plus the *length* of the cdr of that list (recursively), what is the size of a tree?
- A tree is the leaf-count of the car plus the leaf-count of the cdr!

COUNT-LEAVES

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```

EXERCISE

```
(list 1 (list 2 (list 3 4)))


# what is the result of this from the
interpreter?
```

EXERCISE

```
(list 1 (list 2 (list 3 4)))


# what is the result of this from the
interpreter?
```

```
(1 (2 (3 4)))

# Draw a box-and-pointer for this structure

# Draw a tree for this structure
```

OTHER EXERCISES

```
(1 3 (5 7) 9)


# how do we extract 7?
```

OTHER EXERCISES

```
(1 3 (5 7) 9)


# how do we extract 7?
```

```
(cdr (car (cdr (cdr x))))
```

EXERCISES

```
((7))


# How do we extract 7?
```

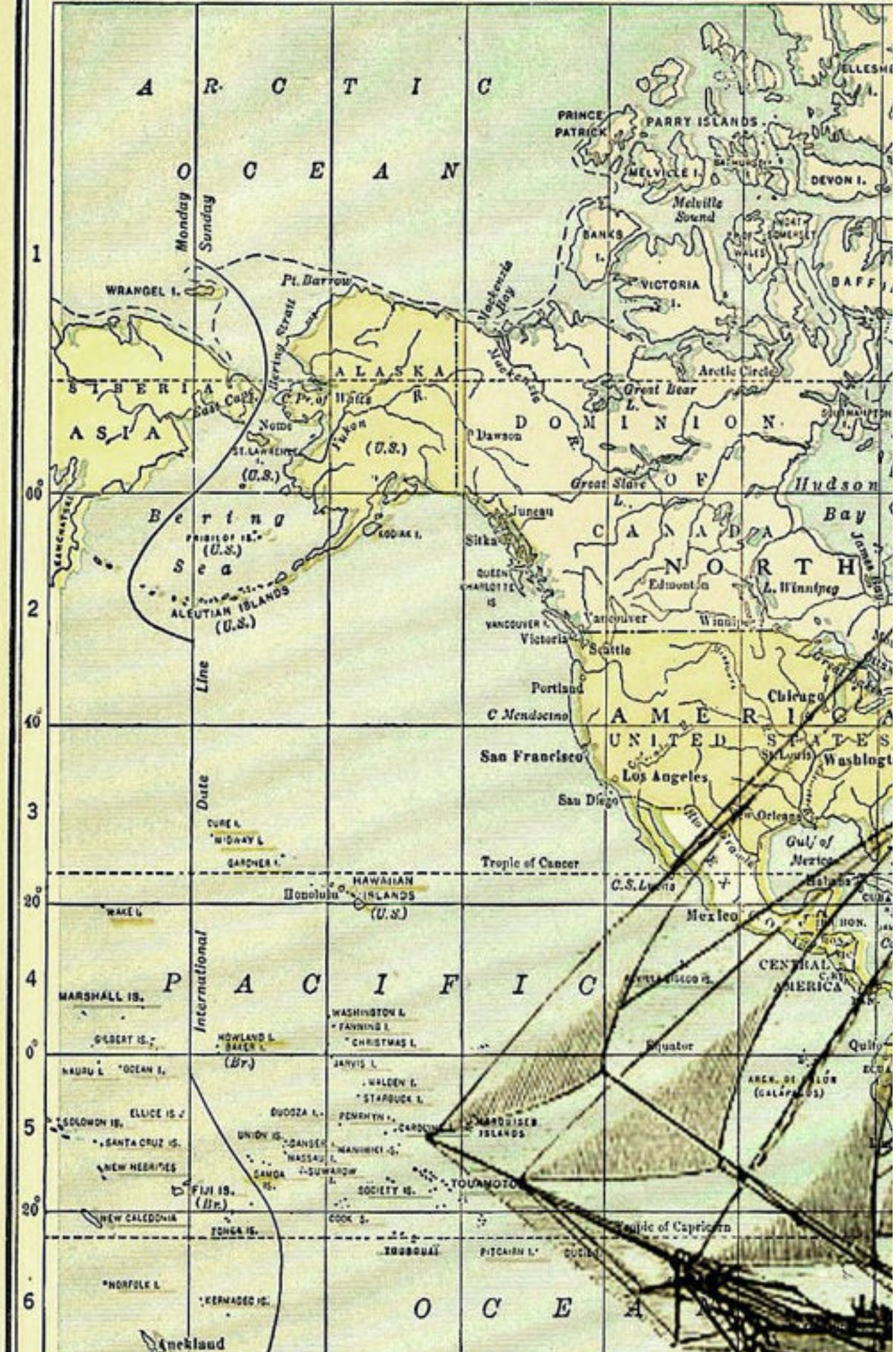
EXERCISES

```
((7))


# How do we extract 7?
```

```
(car (car x))
```

CONVENTIONS WITH SEQUENCES



THE GREAT TRIO

- The conventional interfaces for a sequence consists of three operations: *map, filter,* and *accumulate*
- These should seem familiar; we have worked with them in Rust already, and you may have used them before

WE WON'T BUILD THESE NOW

- It's not appropriate to build filter & accumulate right now
- However, you should ensure that you understand these conceptually!

MAP / FILTER / ACCUMULATE

- Map will always return a sequence of identical size
- Filter will return a sequence of the same or smaller size
- Accumulate will return a single value

HOMEWORK

HOMEWORK 7

In Scheme (you need not test this):

Create the following structure using list:

(1 (2 (3 (4 (5 (6 7))))))

Provide the correct sequence of car and cdr commands to extract the value '7' from this list

Just for fun: Favorite programming blog / podcast / publication?