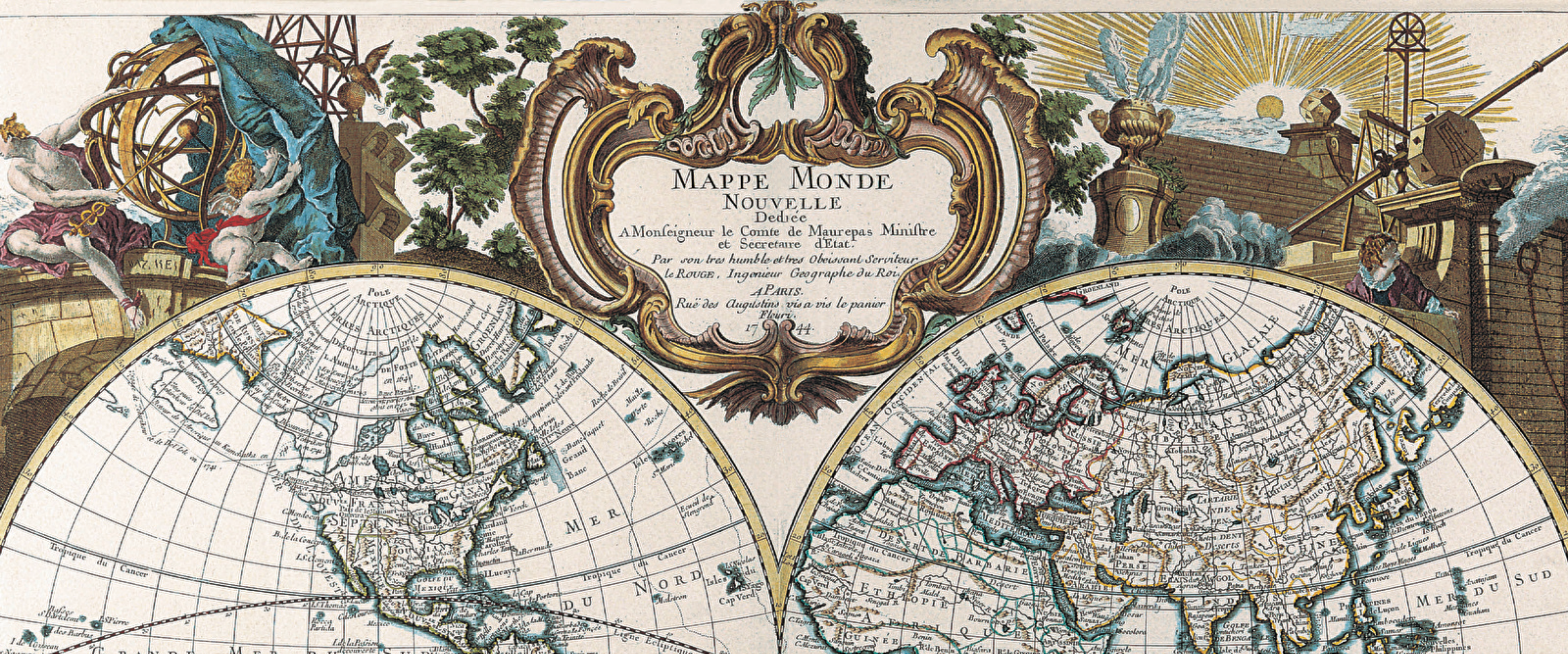
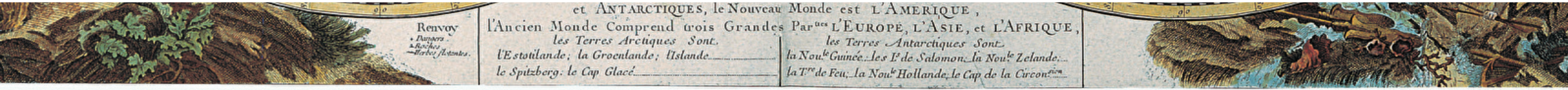
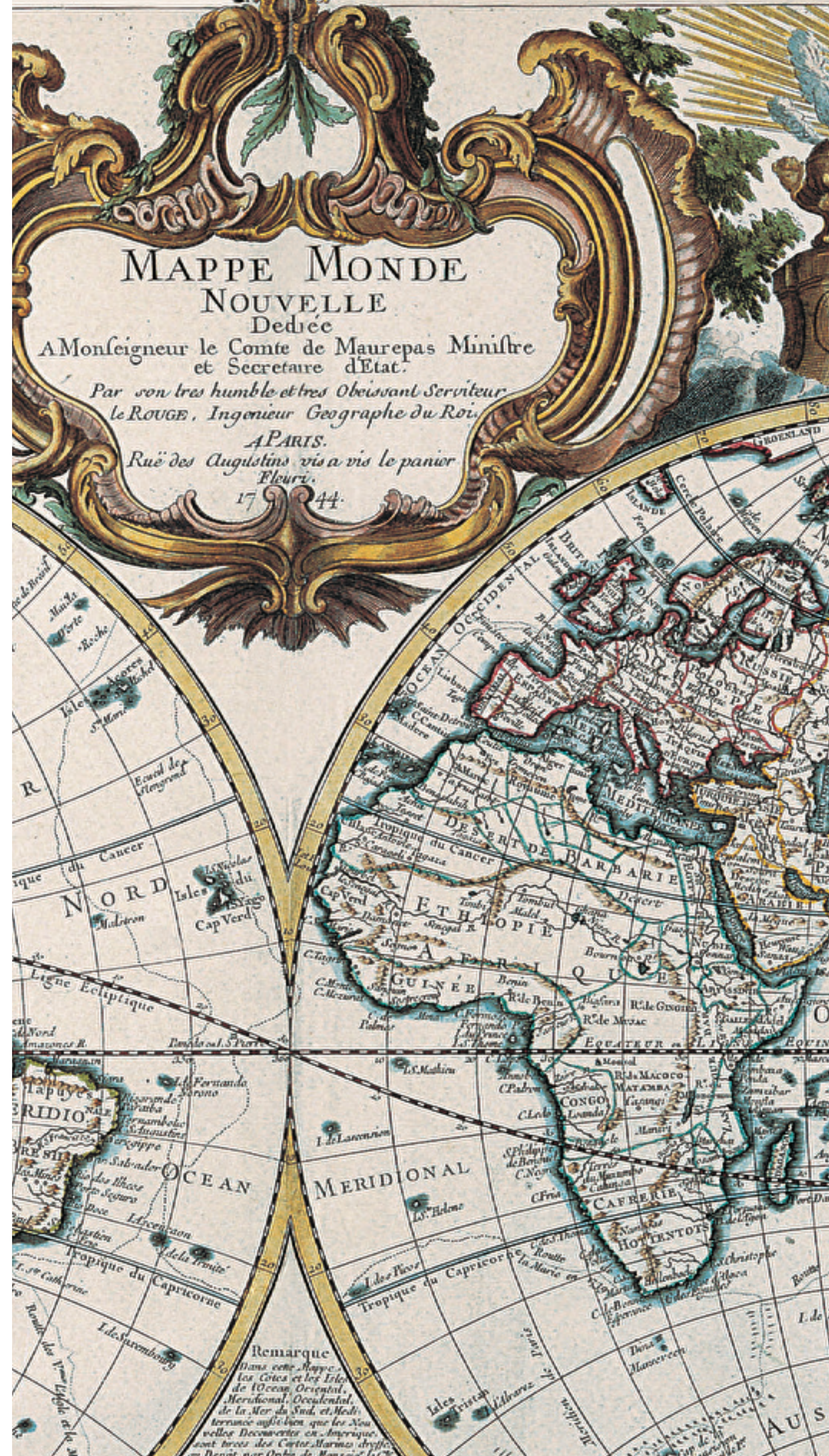


DATA ABSTRACTIONS

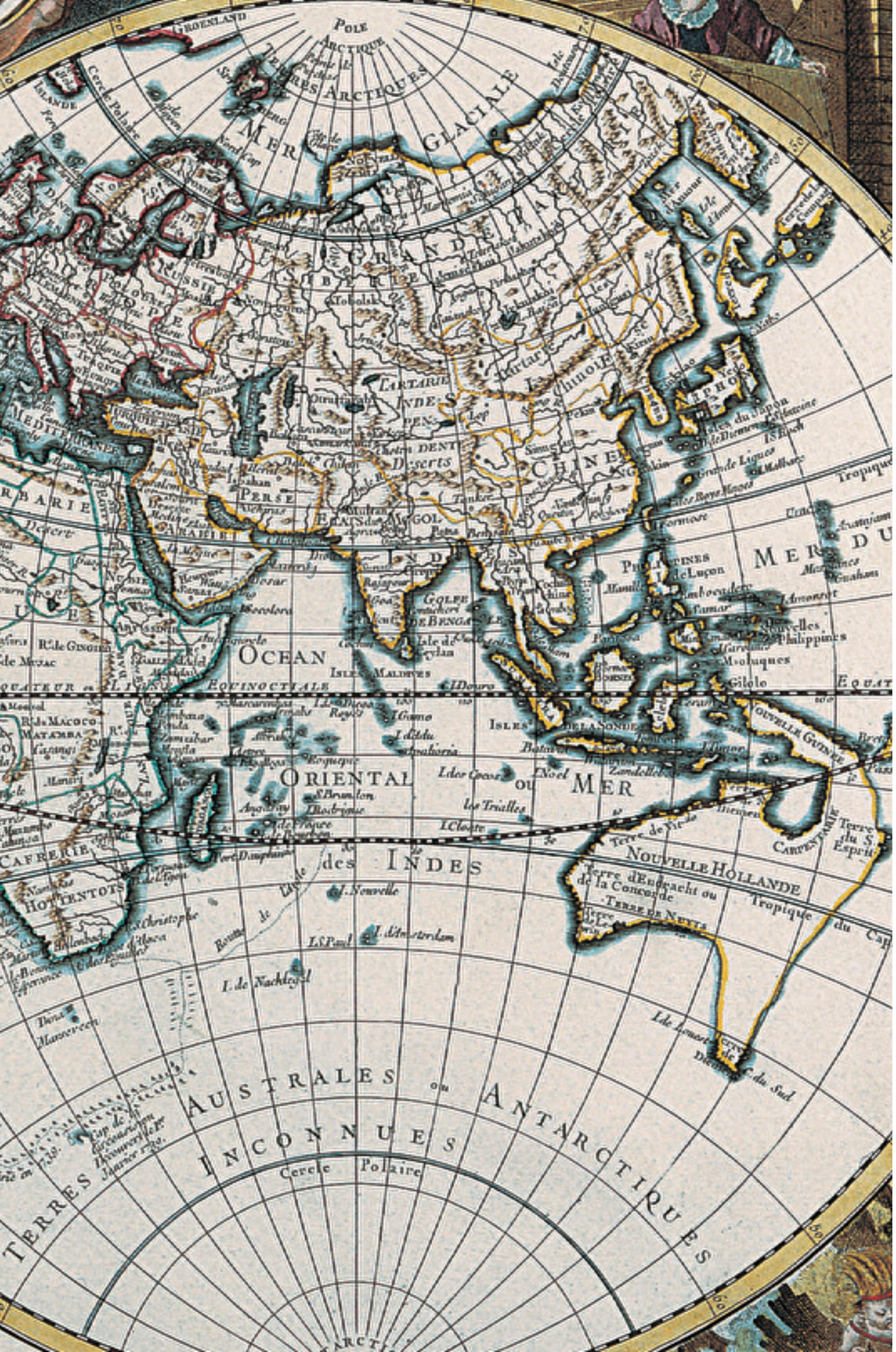




HOMEWORK REVIEW

For the brave

TIMEBOX



TRICKY HOMEWORK

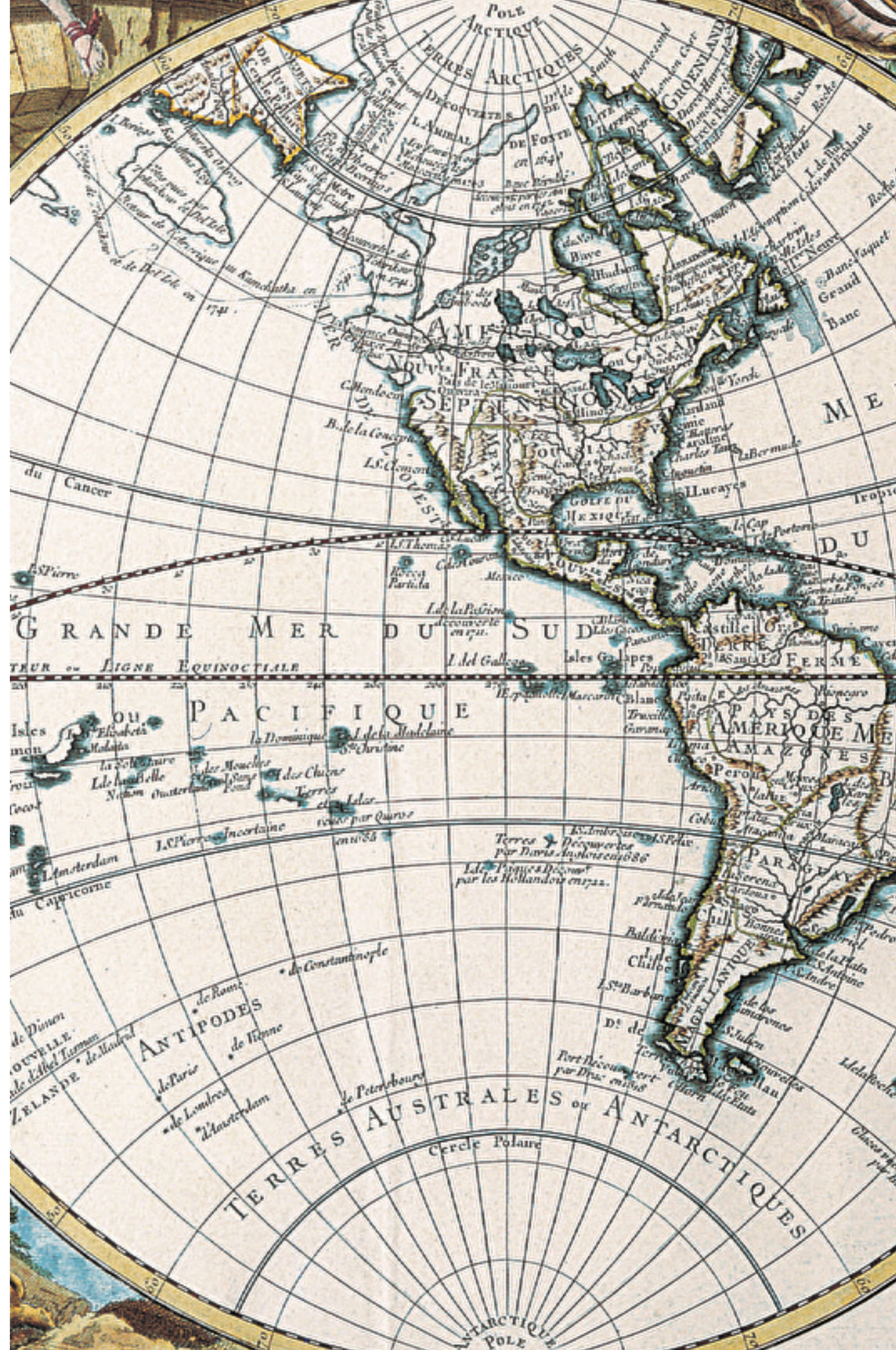
- So for starters, I do sort of apologize for giving you homework on the weekend around an exam
- Second, some of y'all thought too hard about this, or not hard enough
- I saw *so many different attempts* to make this work!
- And *so many didn't work!*
- And some of y'all didn't say "my code doesn't work"…

SUBTRACTION — MY WAY

```
fn evaluate(array: Vec<Primitive>) -> i32 {
    let element = &array[0];
    let mut iter = array.iter();
    iter.next();
    match element {
       Primitive::Subtract => {
            let start = iter.next();
            if let Some(Primitive::Number(val)) = start {
                iter.fold(*val, |total, next|
                    total - evaluate(vec![*next]))
            } else {
                0
            }},
        Primitive::Number(val) => *val
    }
} // Add and multiply not shown…
```

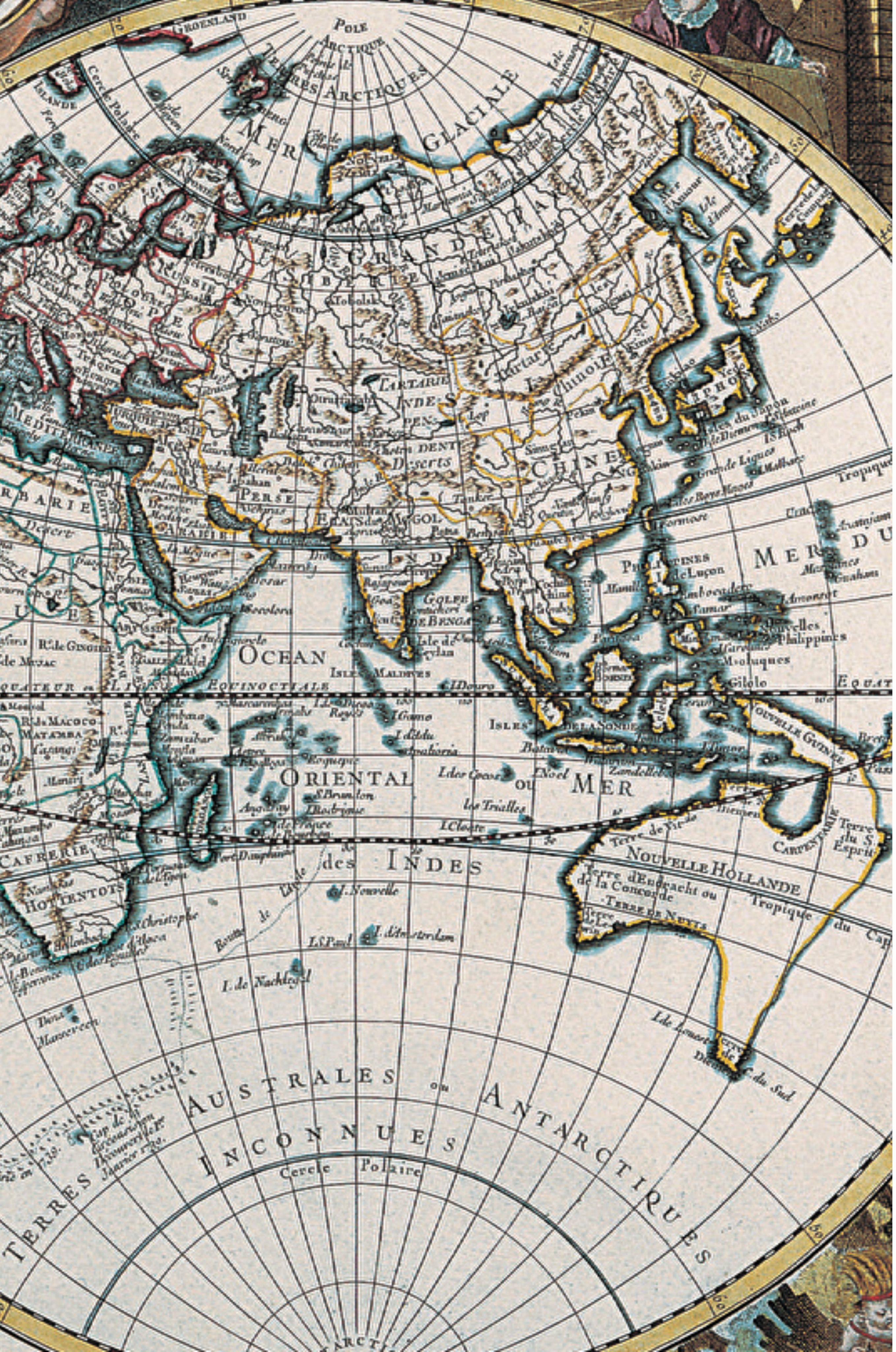
THERE MAY BE SIMPLER WAYS

- I might have liked something that called add recursively, or something like that
- Or perhaps multiplying each element other than the first by negative one via a map() (that would have been cool…)



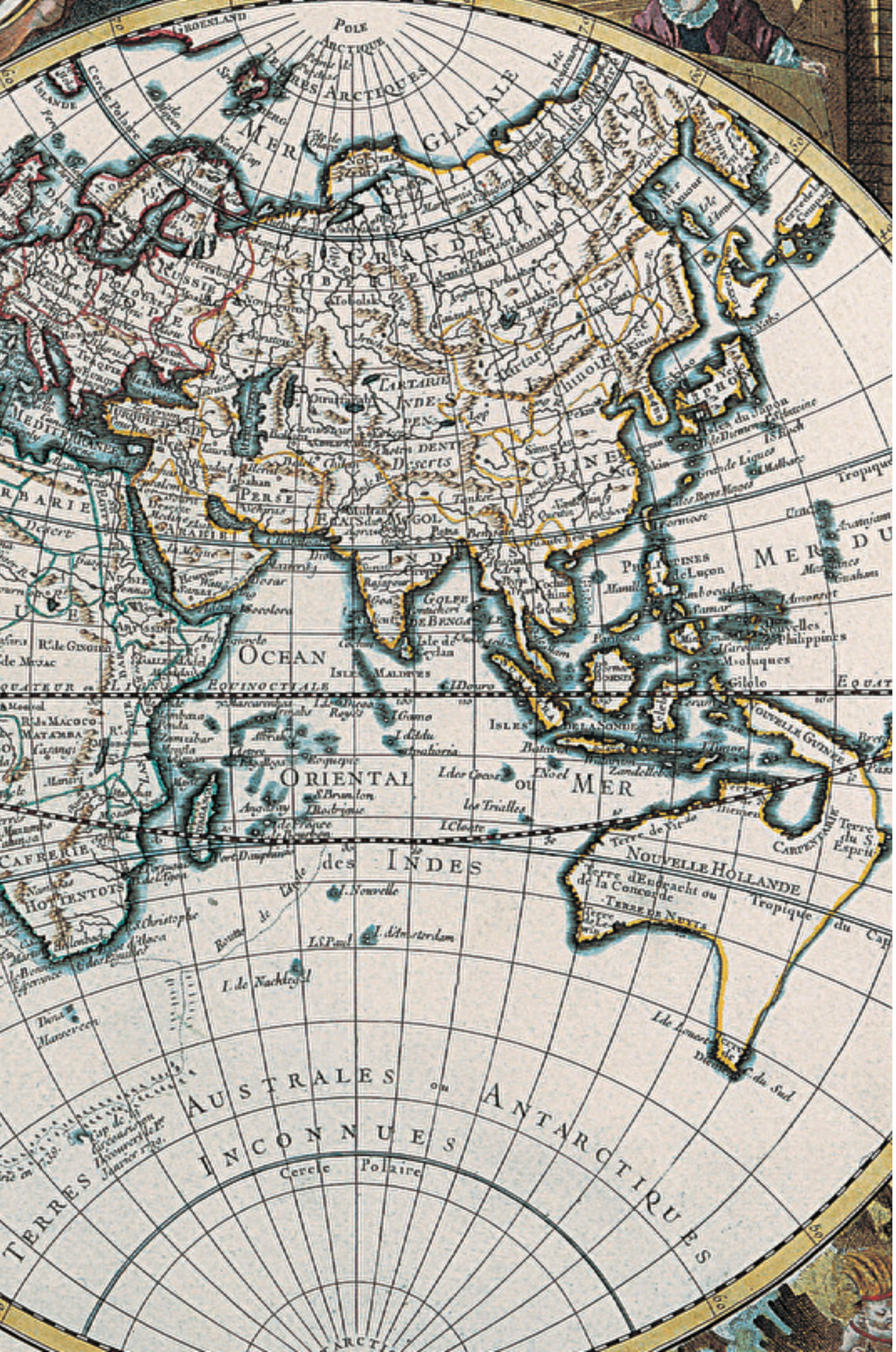
SURE, THIS IS COOL…

```
Primitive::Subtract => {
    let start = iter.next();
    let others = iter.map(|x| evaluate(vec![*x]) * -1);
    if let Some(Primitive::Number(val)) = start {
        others.fold(*val, |total, next| total + next)
    } else {
        0
    }
},
```



UNARY MINUS

- All of this is in support of *subtraction*
- Subtraction requires two operands
- It is not meant to evaluate a negative number — that is a *unary* operator
- (- 1) is not the same thing as (-1)
- Determining subtraction vs unary minus is not the responsibility of the *evaluator*, but rather the *parser*

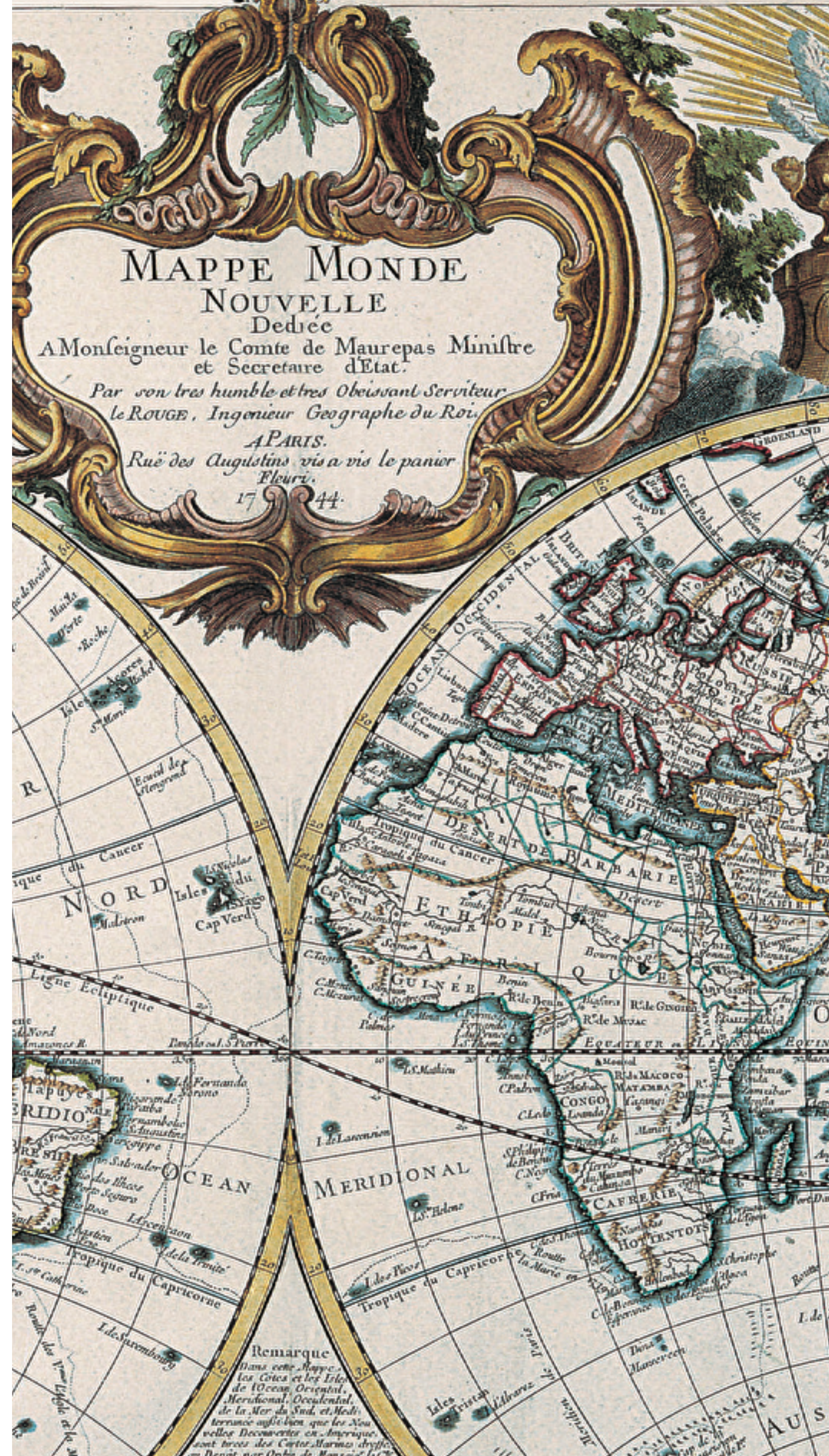


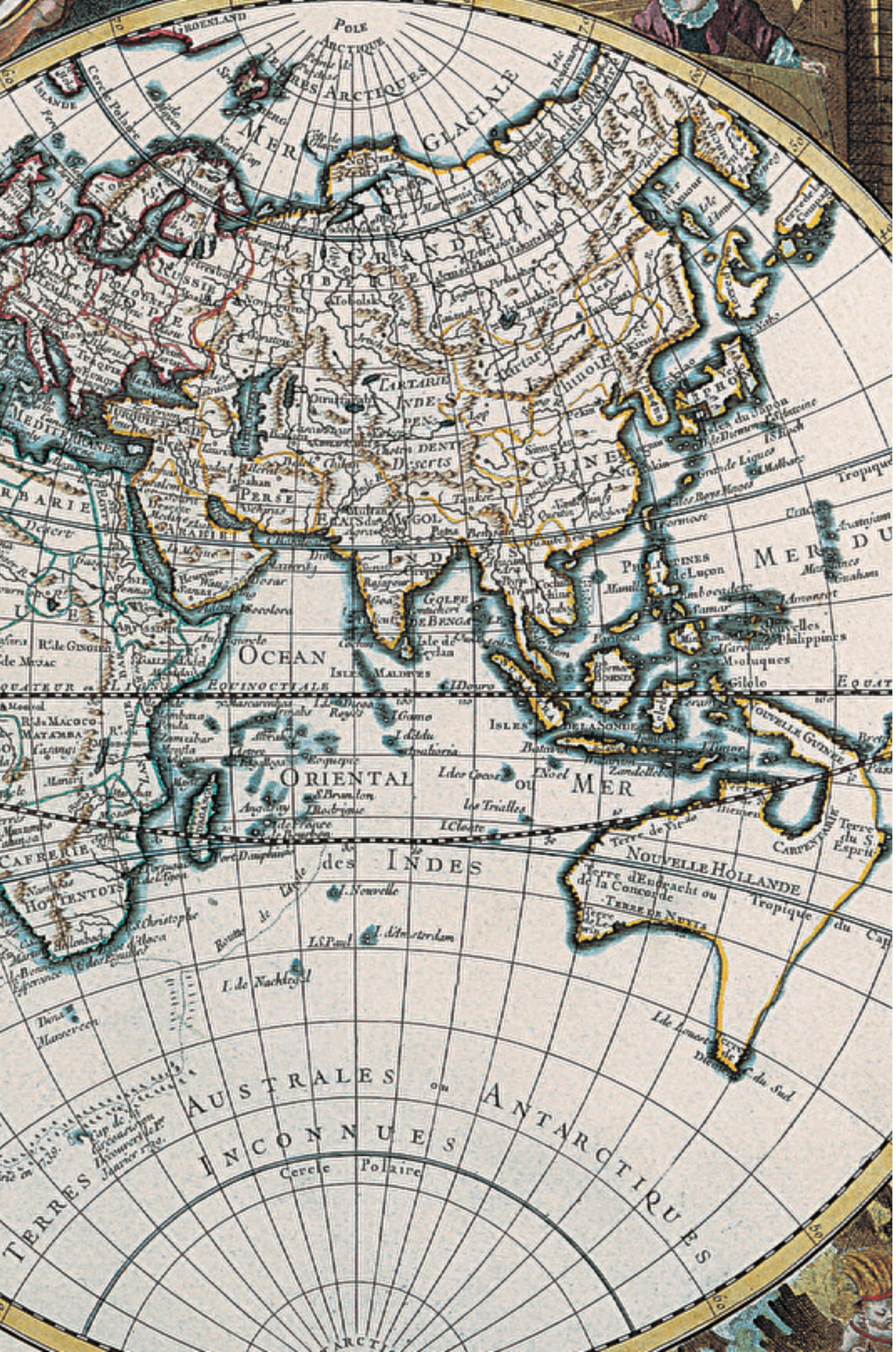
MORE THAN ONE WAY

- All of that to say there's more than one way to complete the task
- Testing is important; next week we'll talk about adding unit testing to our Rust code

COMPOUND DATA

Wishful thinking, layers of abstraction



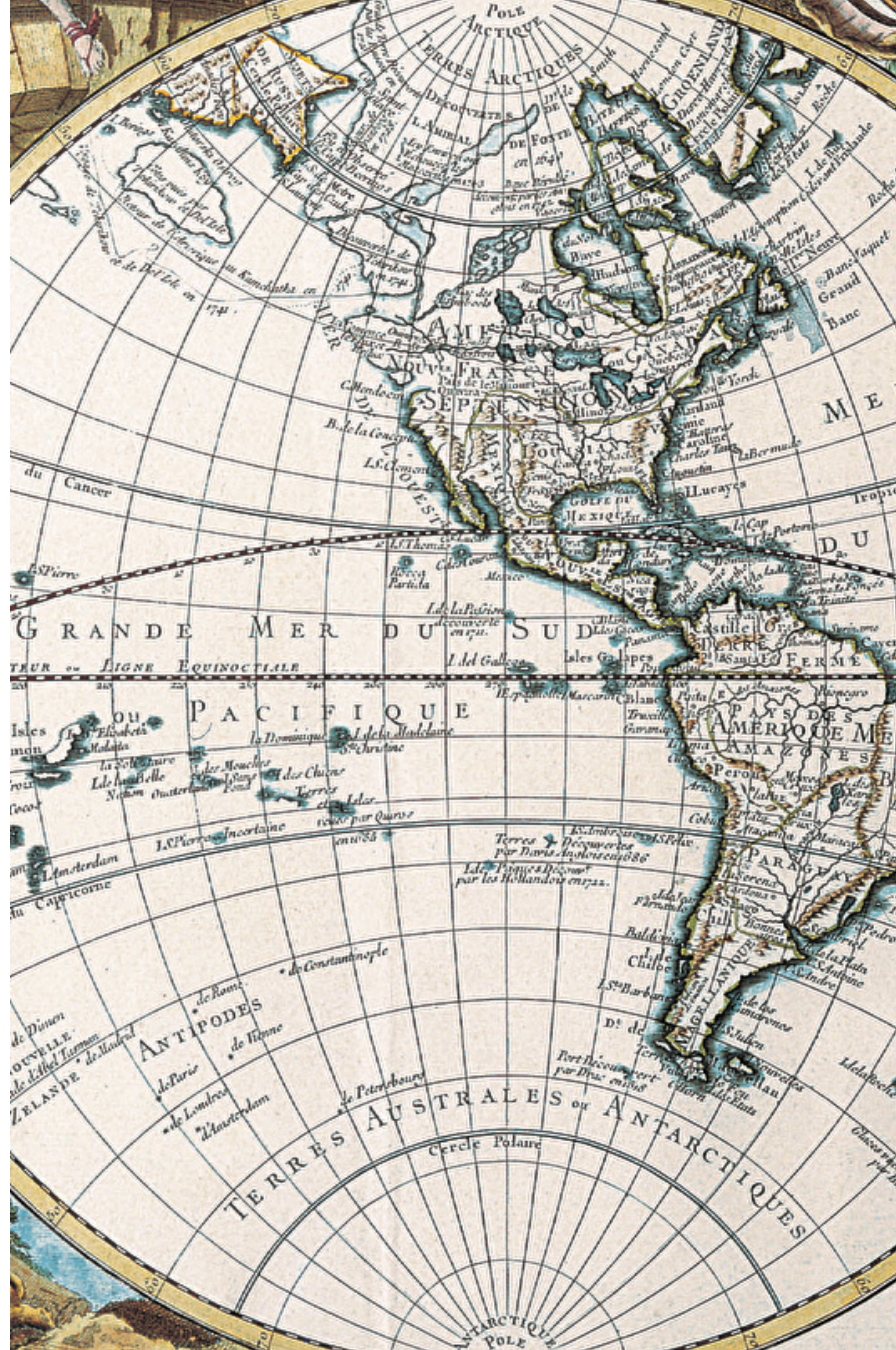


ON TO DATA

- During the first part of this semester, we spent time talking about procedures
- Procedures are only one portion of abstraction
- By moving from simple data into more complex data, we can model more complex (closer to real-world) problems

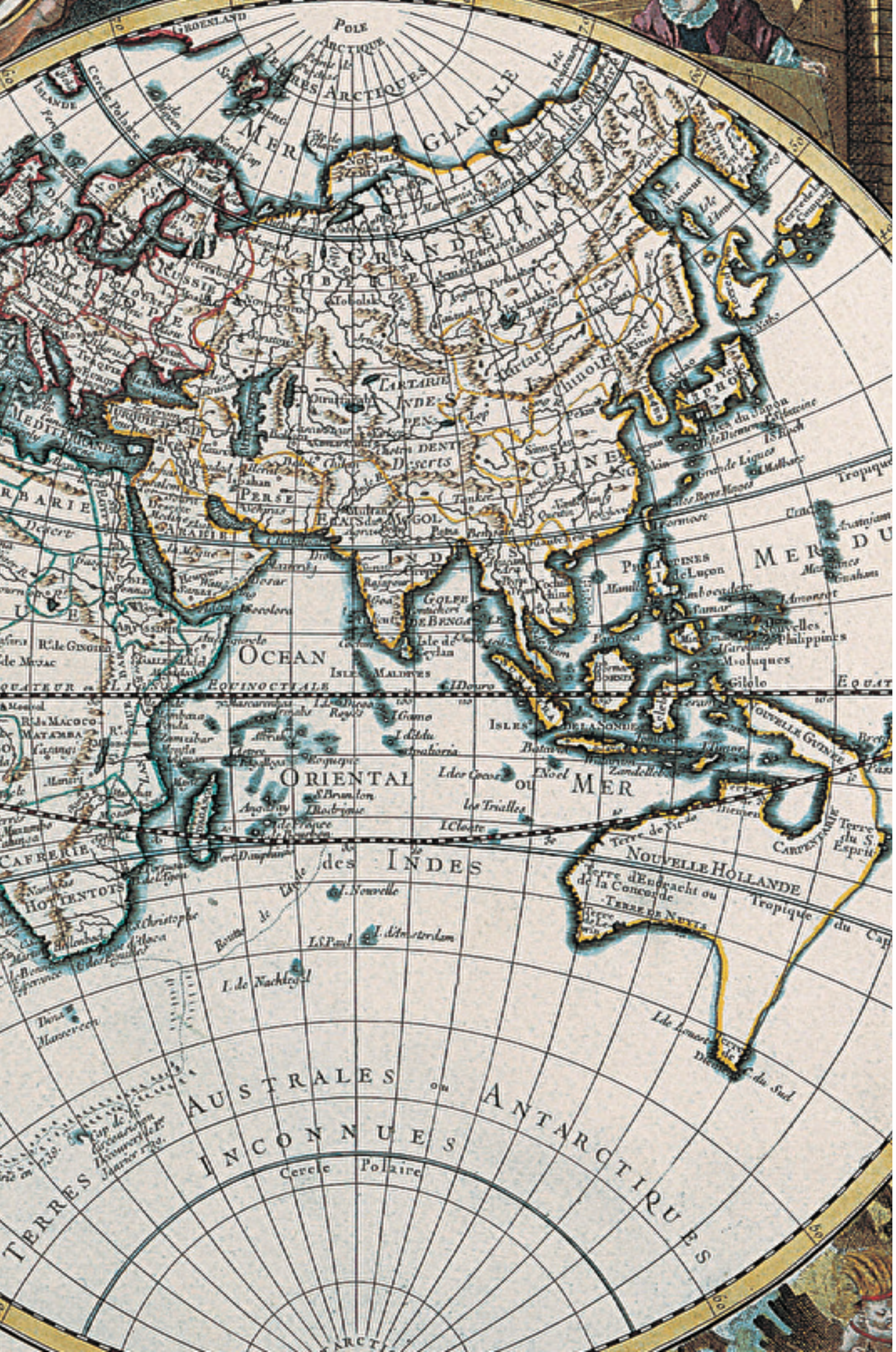
DATA ABSTRACTION

- Use compound data objects
- Programs should work on "abstract data"
- Program should make no more assumptions about the data than the absolute minimum to accomplish its tasks
- Concrete data representation is defined separate from the program that uses it
- Interface between the two
  - Constructors
  - Selectors



RATIONAL NUMBERS

- Sorry, I keep saying "I don’t do math", and then here we go again…
- While we can *use* floating point numbers to approximate rational numbers, we know that they aren’t *exactly* the same thing
- Problems can come in when we start deviating from a ULP of one
  - Unit of least precision (won’t be on a test, I promise)
- Let’s define addition and multiplication of rational numbers



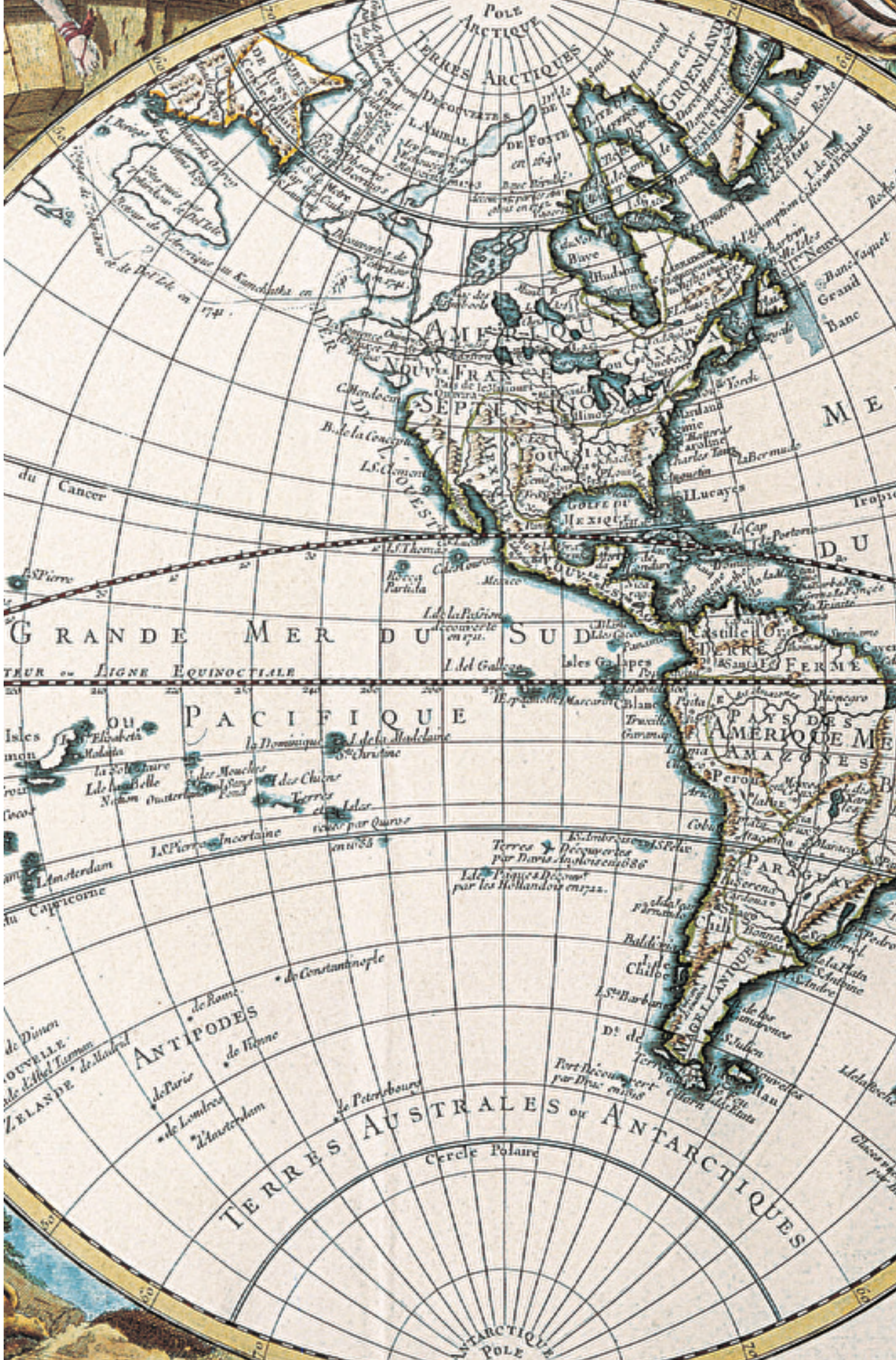
RATIONAL NUMBER

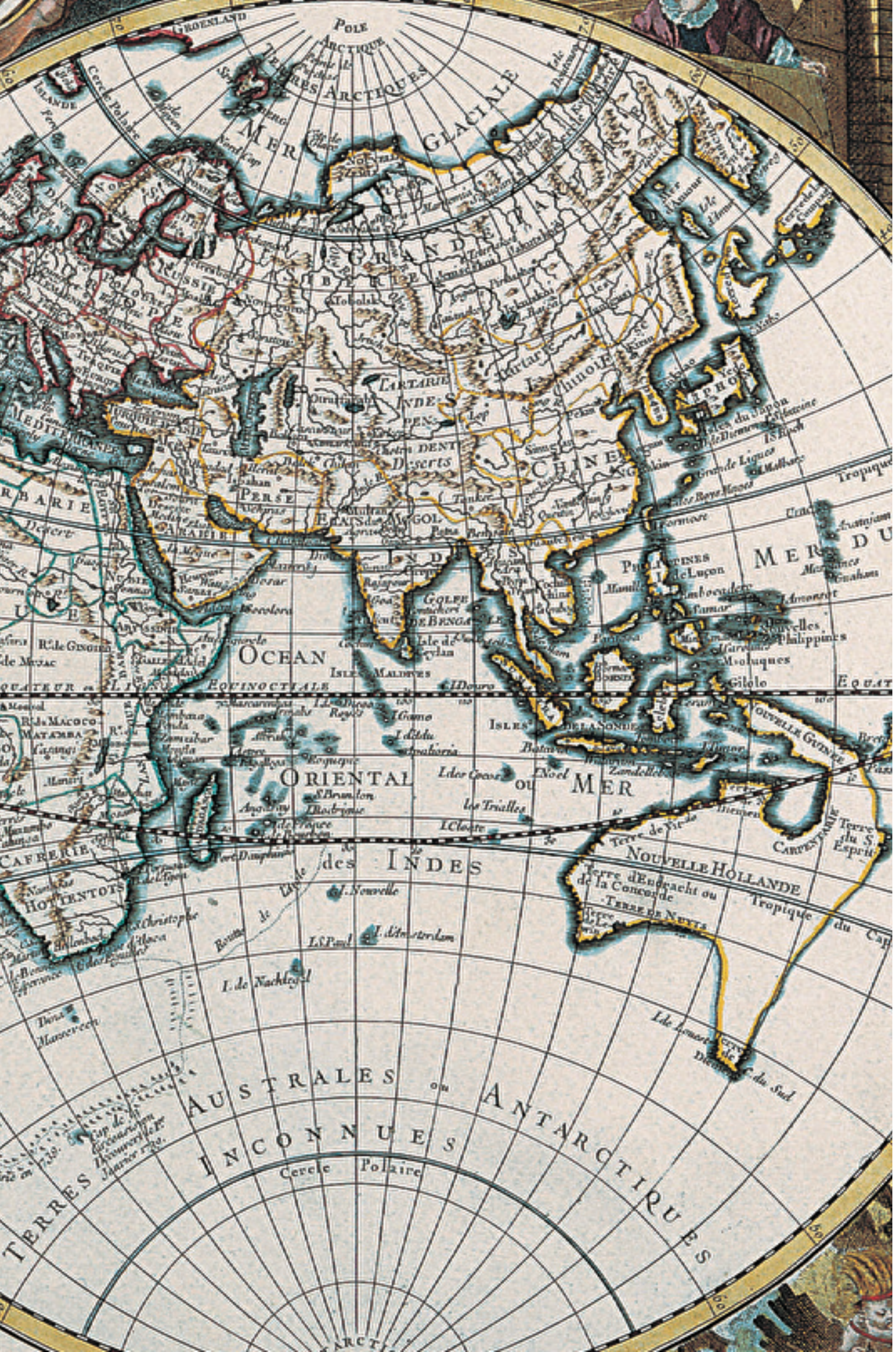
- A rational number is a number that is comprised of a *numerator* and a *denominator*
- We can then imagine an interface to a program that works with numerators and denominators

```
(make-rat <n> <d>)
(numer <x>)
(denom <x>)
```

CONSTRUCTOR

- The first of these is a *constructor*: given a numerator and denominator, it returns a *rational number*, whatever that means.
- The second and third are *selectors*: given a *rational number*, whatever that is, these will return a numerator or denominator, respectively
- We have yet to define what a *rational number* actually is…



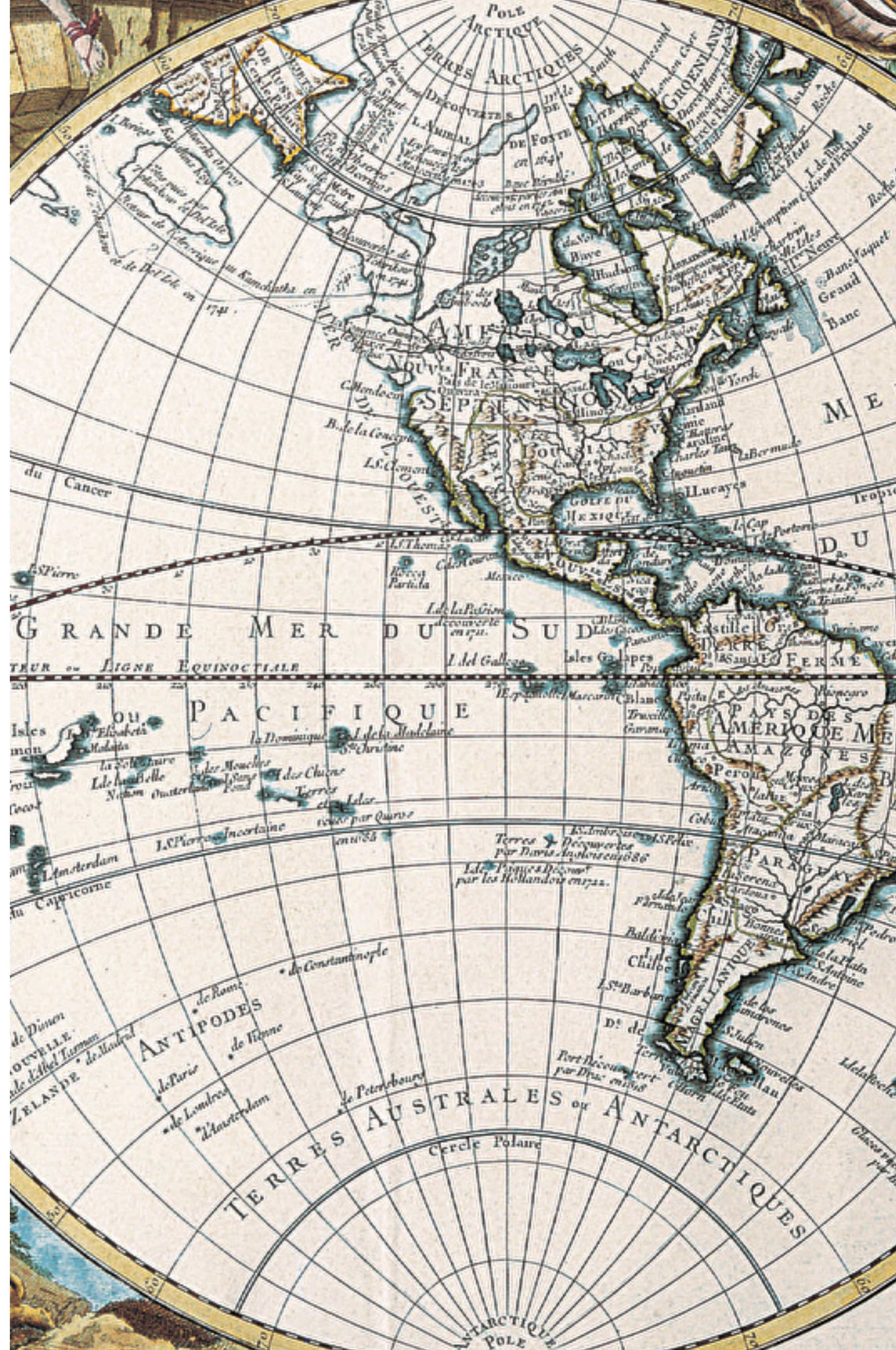


WISHFUL THINKING

- Your textbook (SICP) refers to this as *wishful thinking*, but I do not care for that name
- It's fine to defer definition to a later time, but to wrap this deferral into some kind of magical terminology isn't something I care for
- I prefer the terms *programming by contract* or *protocol-oriented* programming

~~WISHFUL THINKING~~

- Under programming by contract, a definition is set for how portions of a program (whether procedures or data) will behave
- No assumptions about implementation are made beyond the assumption that *it will be implemented at some time by someone*
- I use this technique nearly every day



~~WISHFUL THINKING~~

- By assuming these three procedures, we can perform operations on those rational numbers!

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

ADDING RATIONAL NUMBERS

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
```

ADDING RATIONAL NUMBERS

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
```

- We make no assumptions other than
  - The returned value will be a rational number
  - We can access numerators and denominators
  - We can create a rational number
  - x and y, the formal parameters, are rational numbers

MULTIPLYING RATIONAL NUMBERS

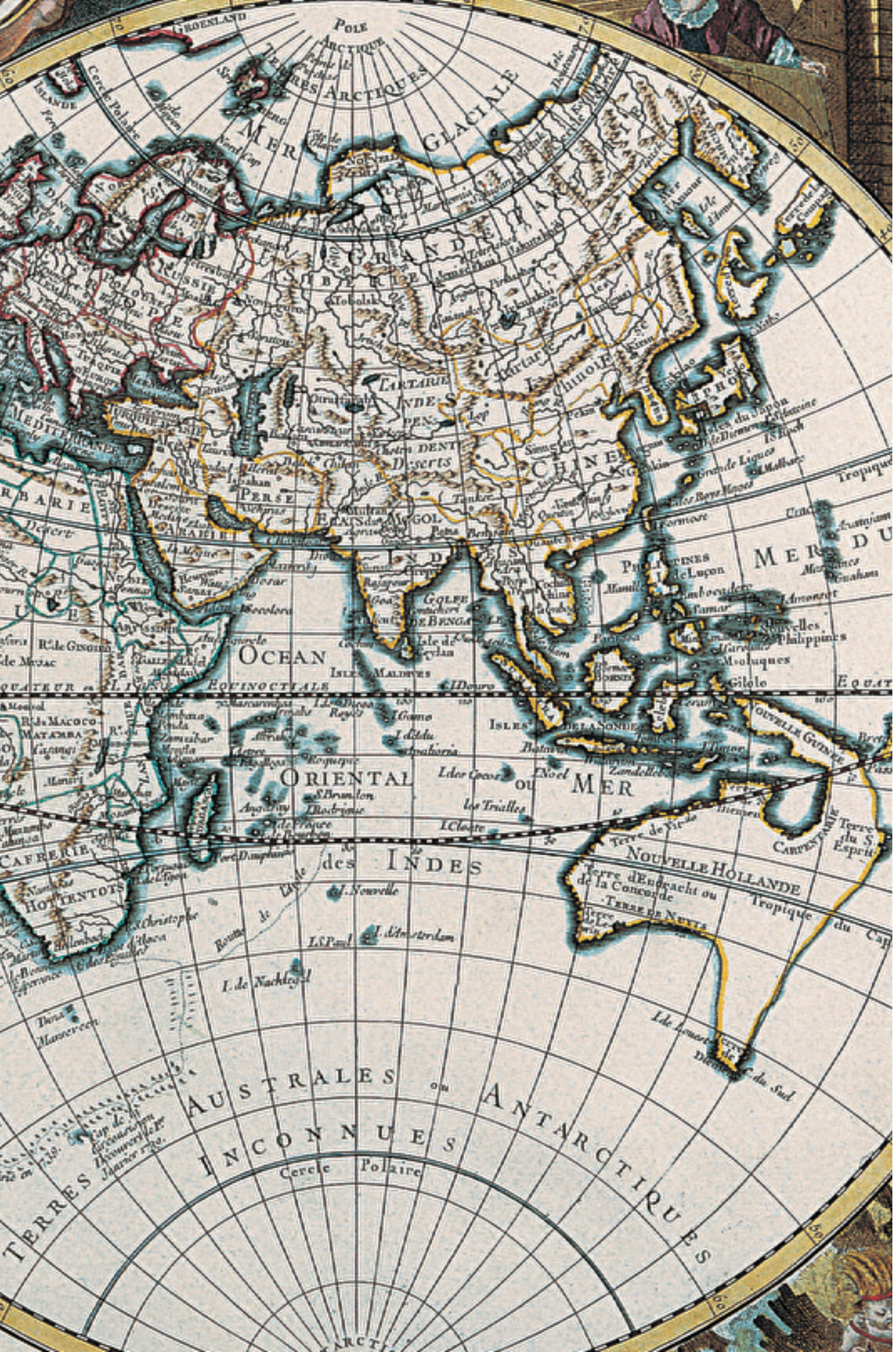
$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

```
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

ACCEPTING THE CONTRACTS

- By accepting the contracts for *make-rat*, *numer,* and *denom,* we can perform the work we wanted to do with rational numbers
- We don't yet have a functional program
  - Still need to implement the above three procedures
- We were able to think about rational numbers *as an entity* rather than get tripped up on implementation details

PAIRS



PAIRS IN SCHEME

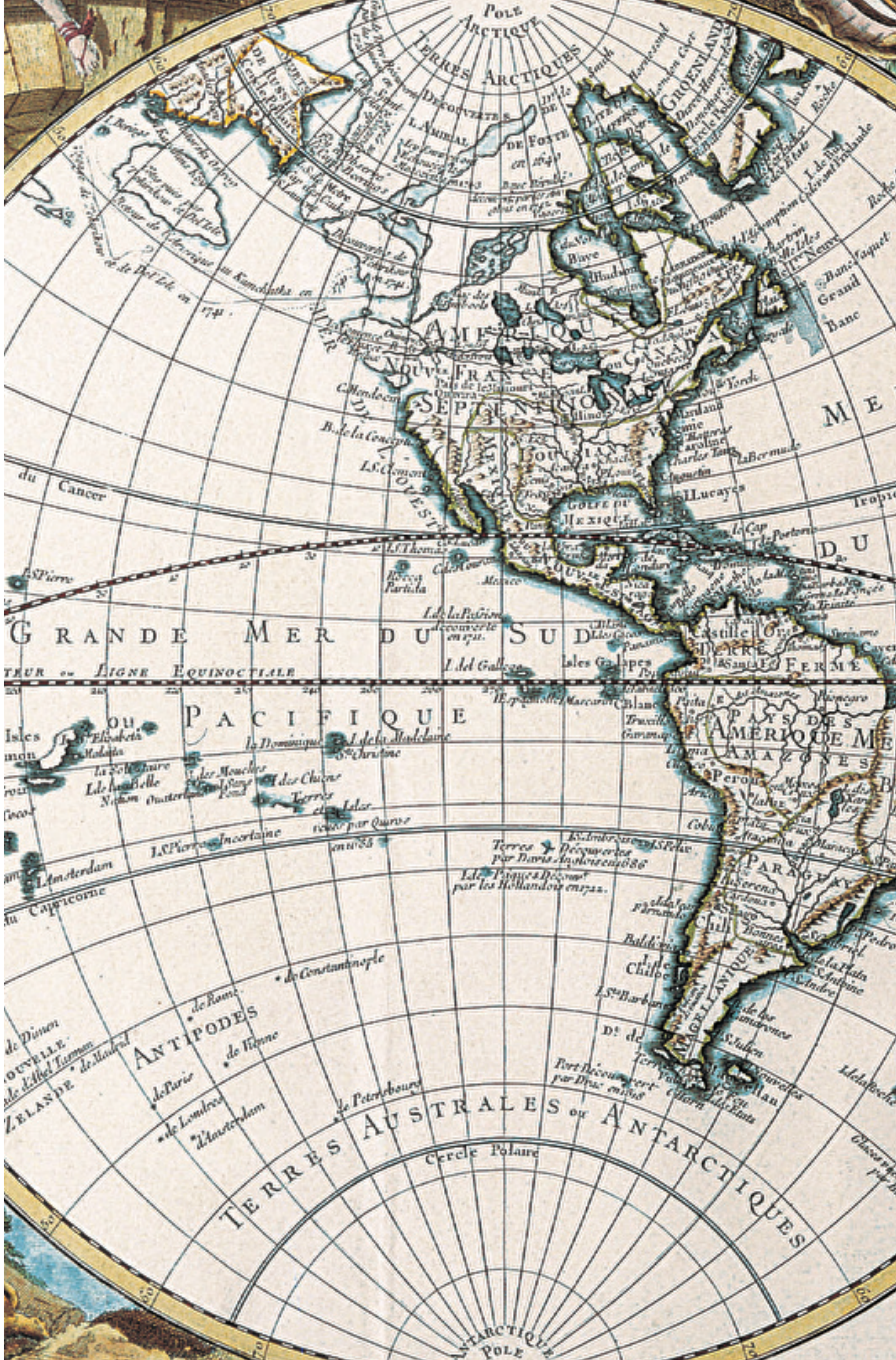
- Scheme provides a *pair*, which consists of two parts
- A *pair* can be constructed with the primitive procedure *cons*
- *Cons* with two arguments will return a compound data object with those two arguments as parts
- Given a pair, we can extract its parts using *car* and *cdr*

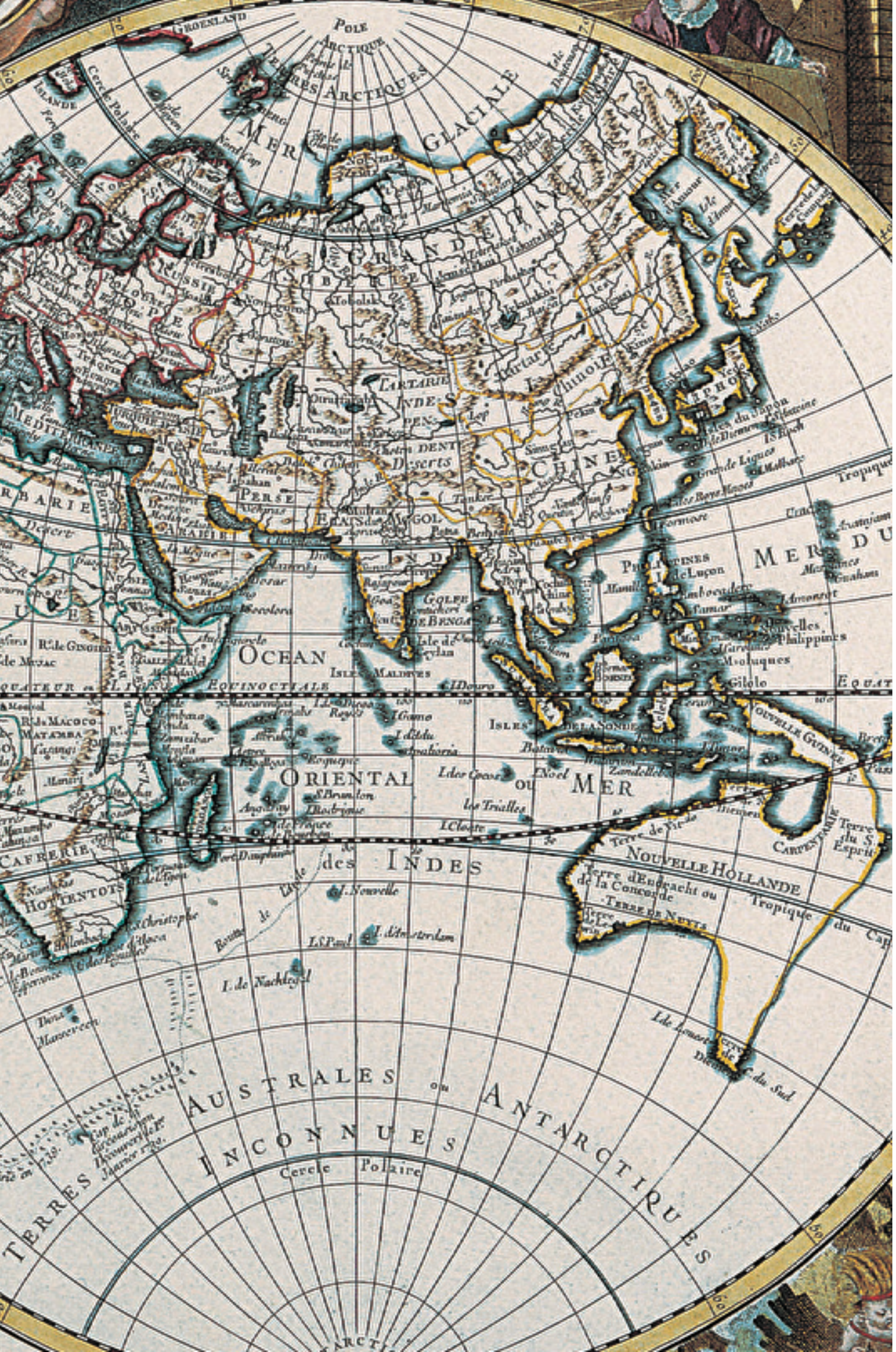
SIMPLE EXAMPLE

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

  x now is a pair consisting of 1 and 2

(car x)

  1

(cdr x)

  2
```

CONS, CAR, CDR

- (cons <a> <b>)
  - this creates a pair from a and b
- (car <x>)
  - this returns the first item added when the pair was created
- (cdr <x>
  - this returns the second item added when the pair was created





NOT JUST SIMPLE DATA

- Pairs are not restricted to containing two simple data elements
- Pairs can also contain other pairs, or null
- Data objects constructed from pairs are *list-structured data*

LIST-STRUCTURED DATA

```
(define x (cons 1 2))
(define y (cons 3 4))
(define z (cons x y))

(car (car z))
    1

(cdr (cdr z))
    4
```

USING LIST-STRUCTURED DATA FOR RATIONAL NUMBERS

```
(define (make-rat n d) (cons n d))

(define (numer x) (car x))

(define (denom x) (cdr x))
```

SUBSTITUTION

```
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))


(define (mul-rat x y)
  (cons (* (car x) (car y))
        (* (cdr x) (cdr y))
```

COMPLETE EXAMPLE

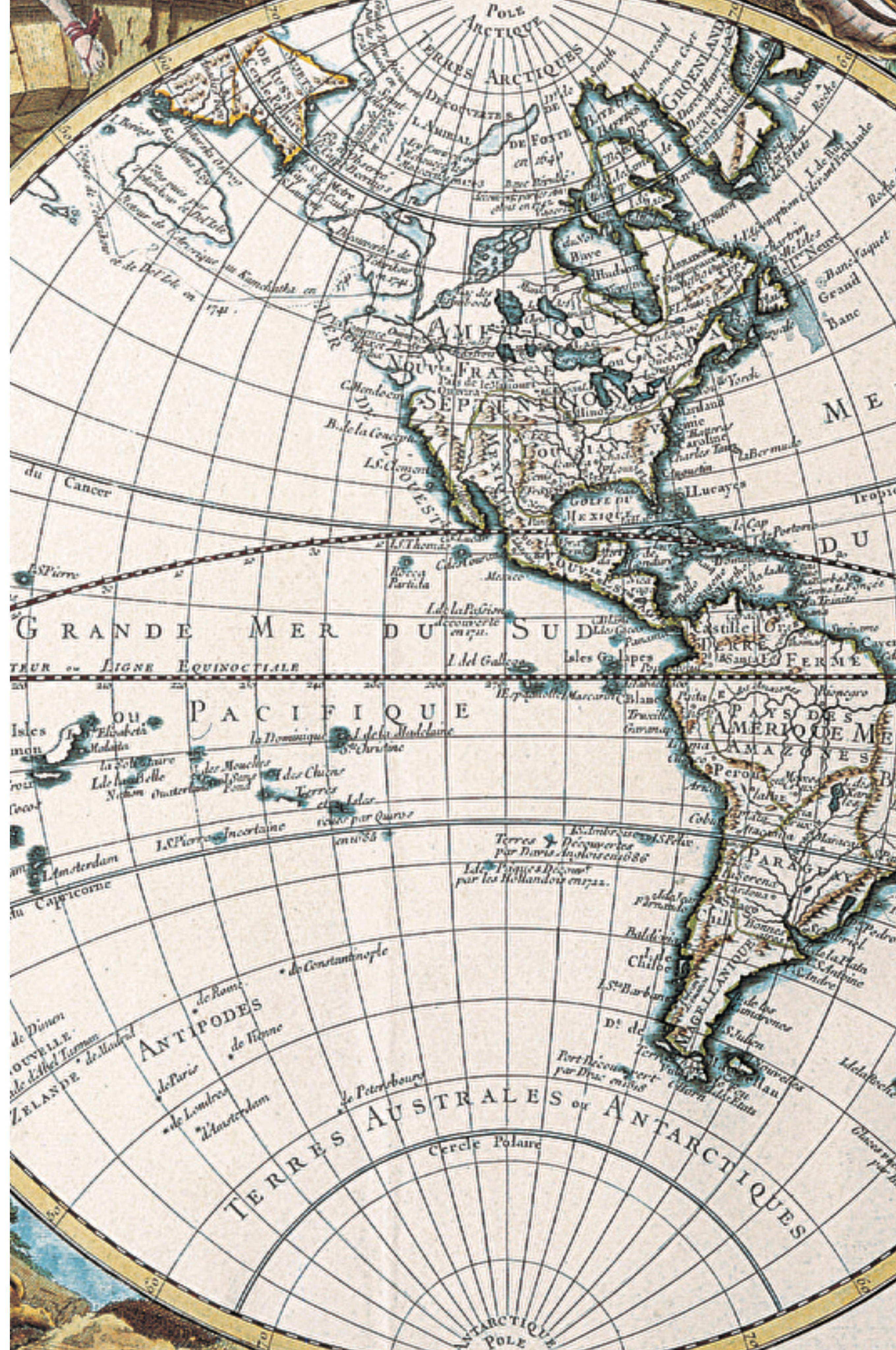
```
(define x (make-rat 1 3))
(define y (make-rat 1 4))
(define z (mul-rat x y))


(define x (cons 1 3))
(define y (cons 1 4))
(define z (cons
           (* (car x) (car y))
           (* (cdr x) (cdr y))))
```

CONTINUING EXPANSION

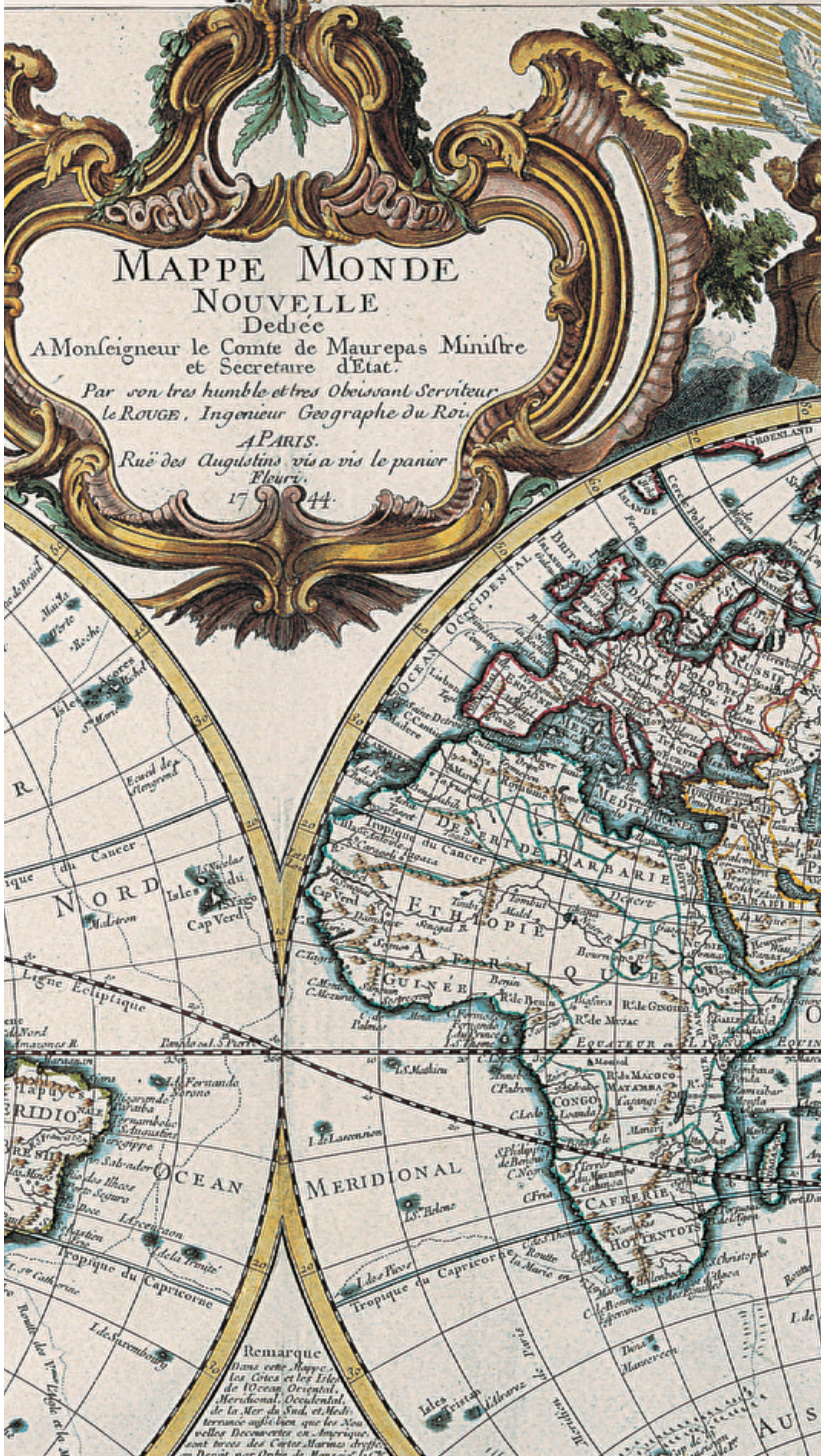
```
(define z (cons
           (* (car x) (car y))
           (* (cdr x) (cdr y))))
(define z (cons
           (* 1 1)
           (* 3 4)))


(define z (cons 1 12))
```

CAR AND CDR

- Just in case you're on *Programmers Jeopardy* some day (or something like that):
- CAR: Contents of the Address Register
- CDR: Contents of the Decrement Register
- These don't mean anything in terms of modern computing hardware, but there you go



ABSTRACTION BARRIERS



ABSTRACTION BARRIERS MODEL

Programs using rational numbers

Rational numbers in problem domain

add-rat, mul-rat, etc

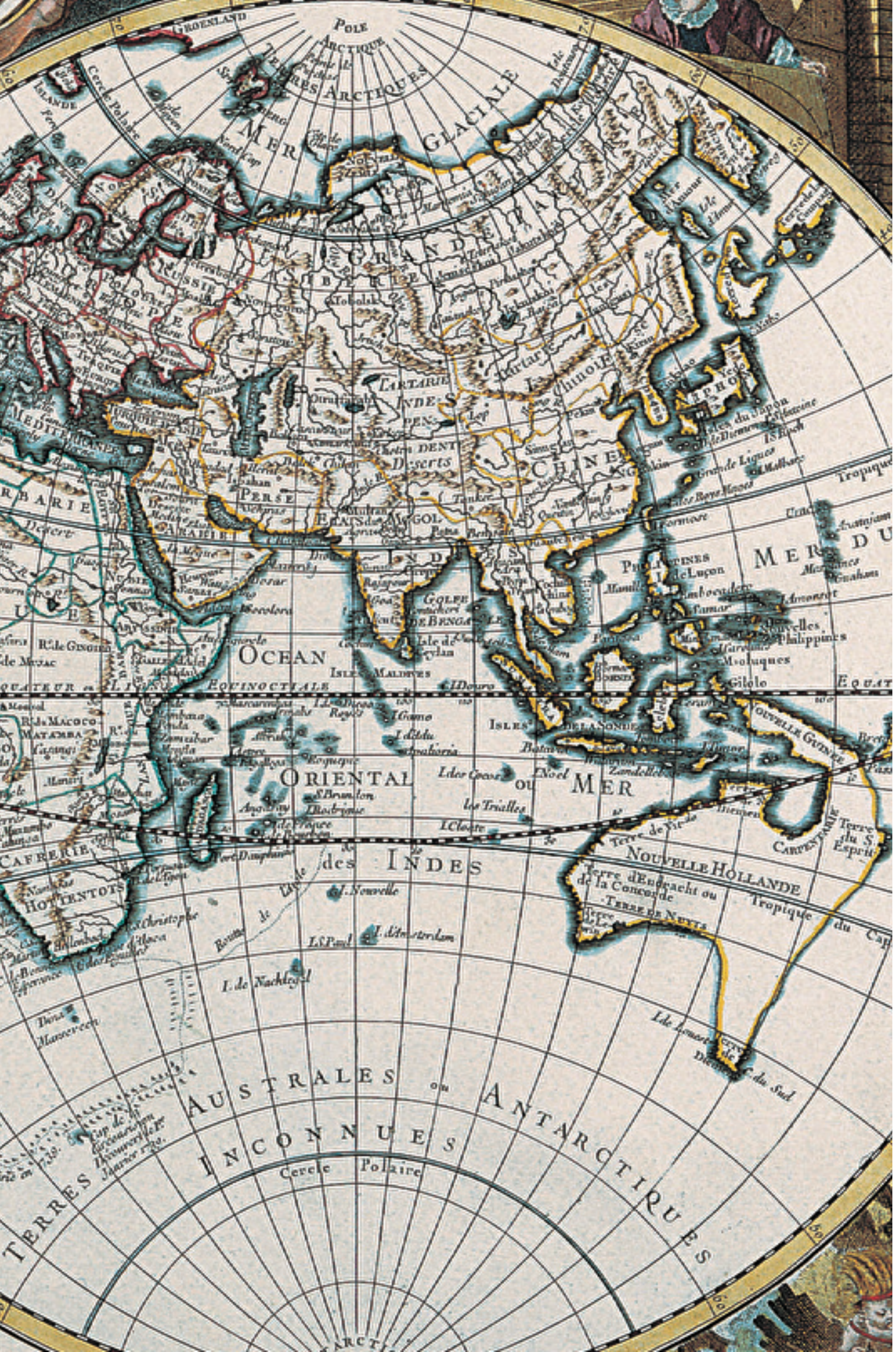
Rational numbers as numerator and denominator

make-rat numer denom

Rational numbers as pairs

cons car cdr

However pairs are implemented

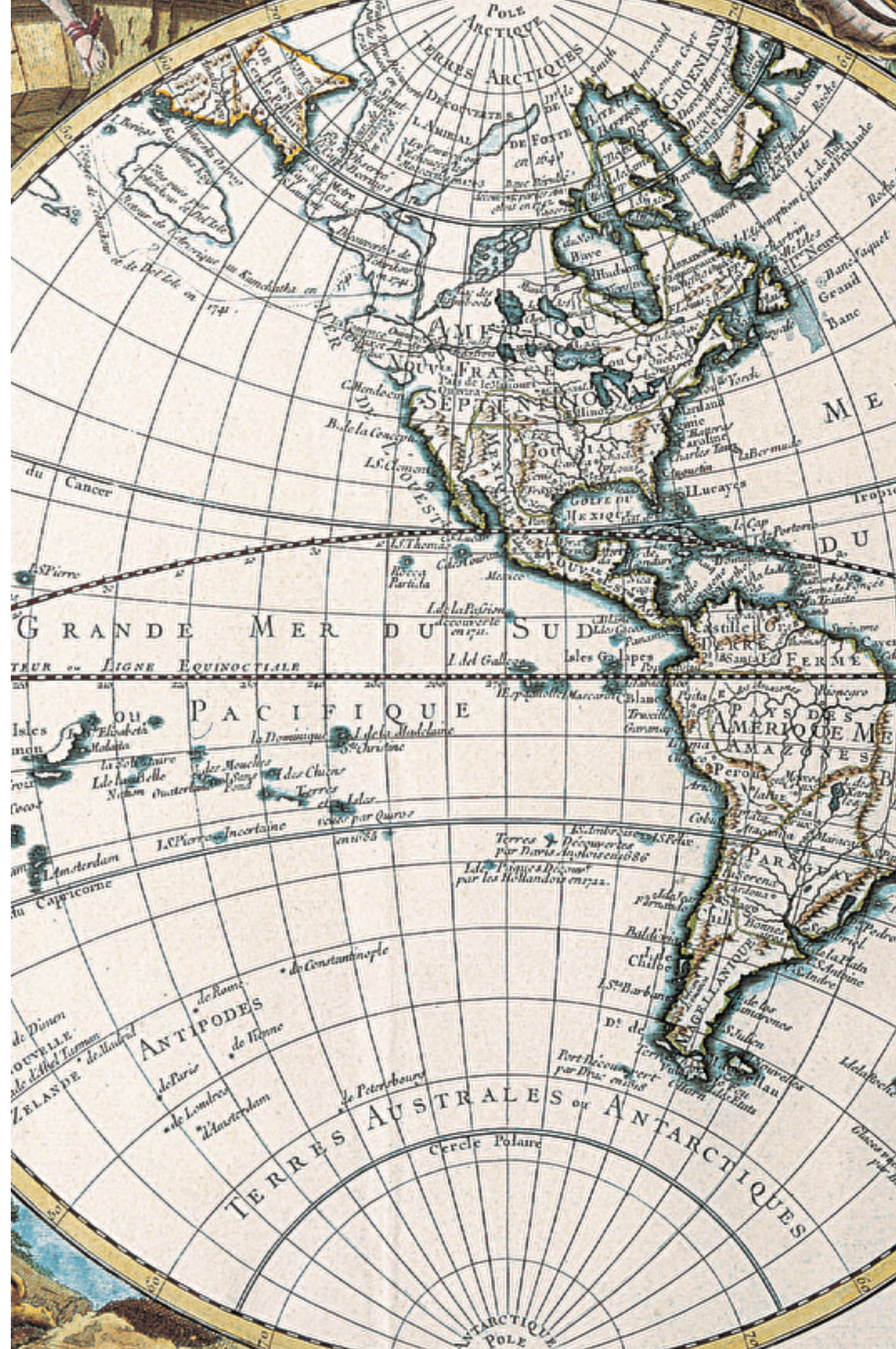


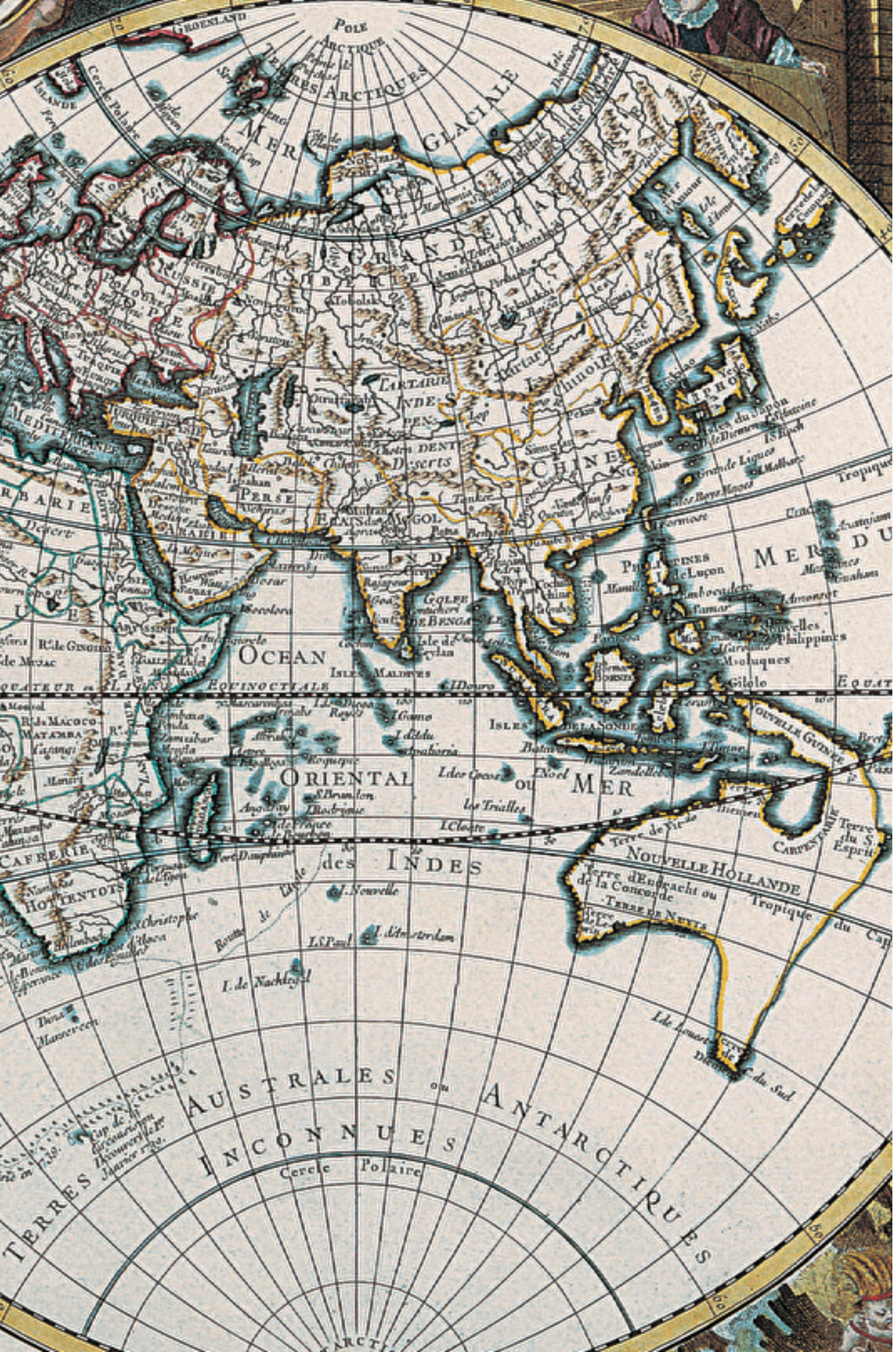
THE ADVANTAGE OF BARRIERS

- Programs that want to use rational numbers strictly concern themselves with the "public use" procedures such as *mul-rat*
- Creating procedures for public use, such as *mul-rat,* does not require knowledge of how the program as a whole might work; rather only *make-rat, numer, and denom*
- The existence of pairs is only relevant to *make-rat, numer, denom*

SEPERATION OF CONCERNS

- A general principle of programming I emphasize is the *separation of concerns*
- Any given portion of a program "knows" as little as possible about other portions of the program
- This may seem a bit extreme given how I’ve walked through all layers in our example thus far, but in a complex, large program— it is the difference between comprehensibility and confusion



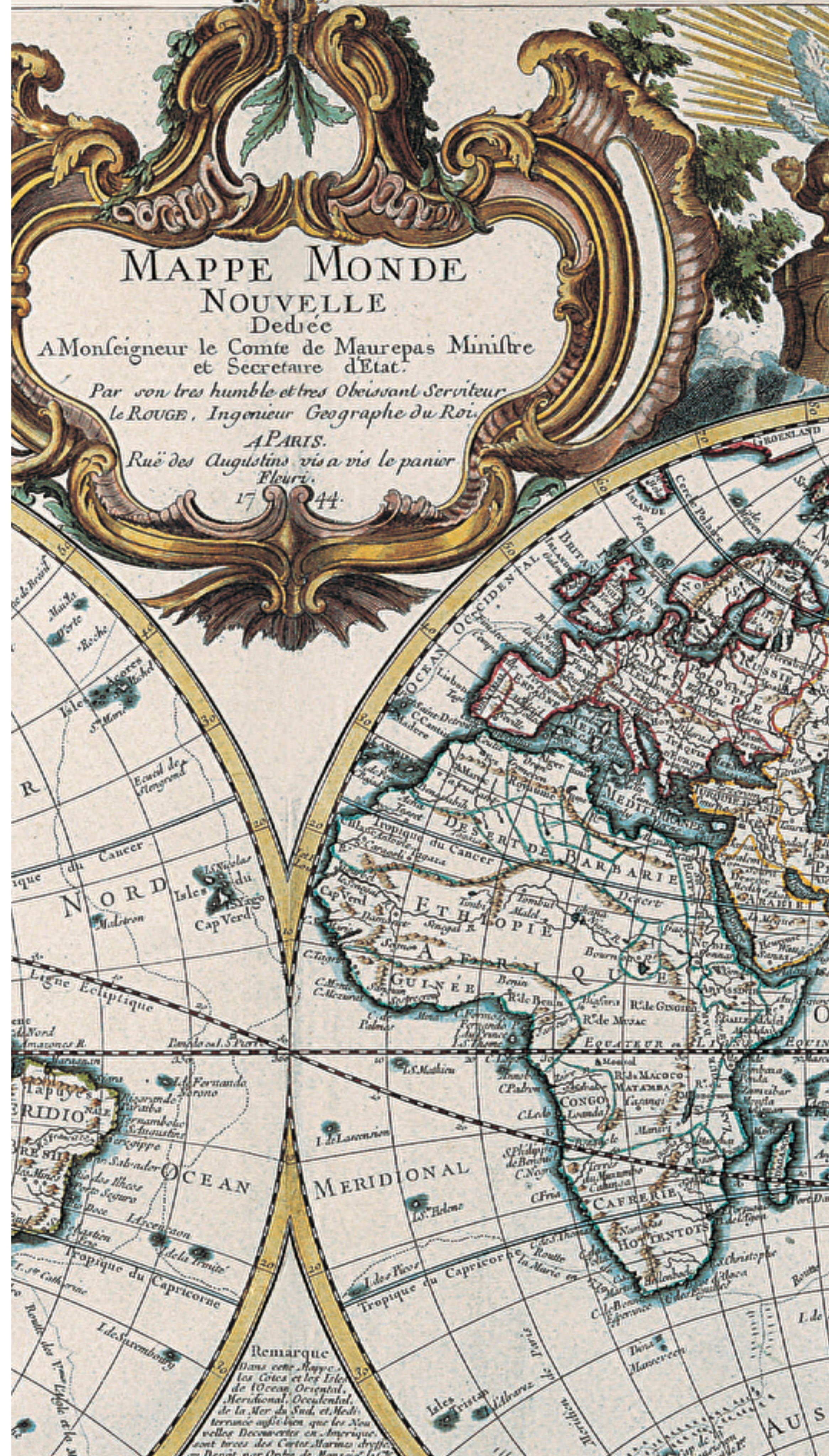


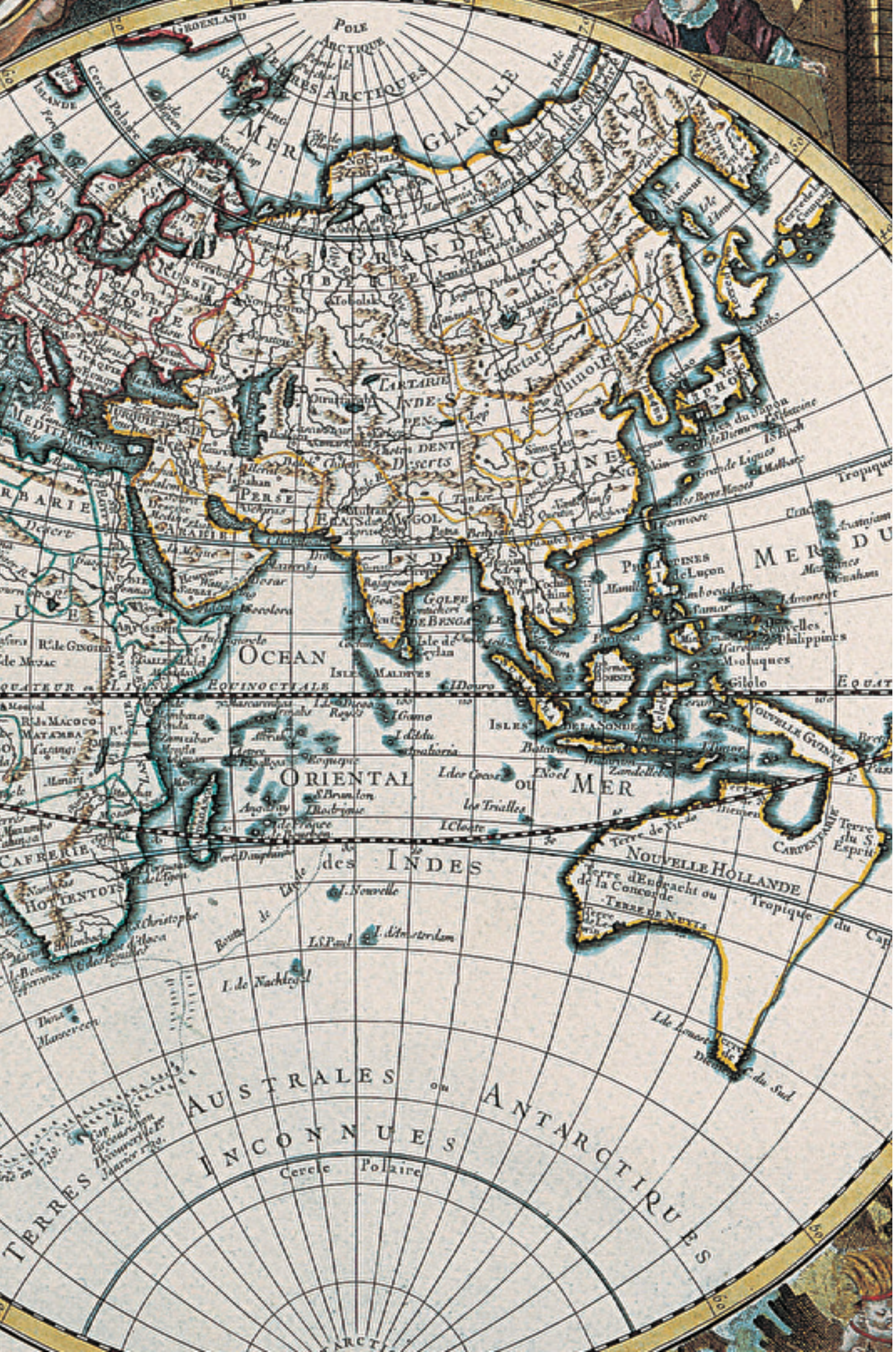
ADVANTAGES

- Modular design
- Independent implementation
  - By different programmers, even!
- Easier to maintain and modify

- Please note this is a *software design* strategy, not a language feature
  - Any language can be programmed in this manner
  - Not restricted to pairs etc.

ANOTHER EXAMPLE: CARTESIAN COORDINATES





CARTESIAN SYSTEM

- Let's create an abstraction for cartesian coordinates
- We'll define points, line segments, and lengths

POINT

```
(define (make-point x y))
(define (point-x a))
(define (point-y a))
```

With this constructor and two selectors, we can build just about any operation we wish for cartesian systems

DISTANCE

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
(define (distance a b)
        (sqrt (+
              (square (- (point-x b) (point-x a)))
              (square (- (point-y b) (point-y a))))))
```

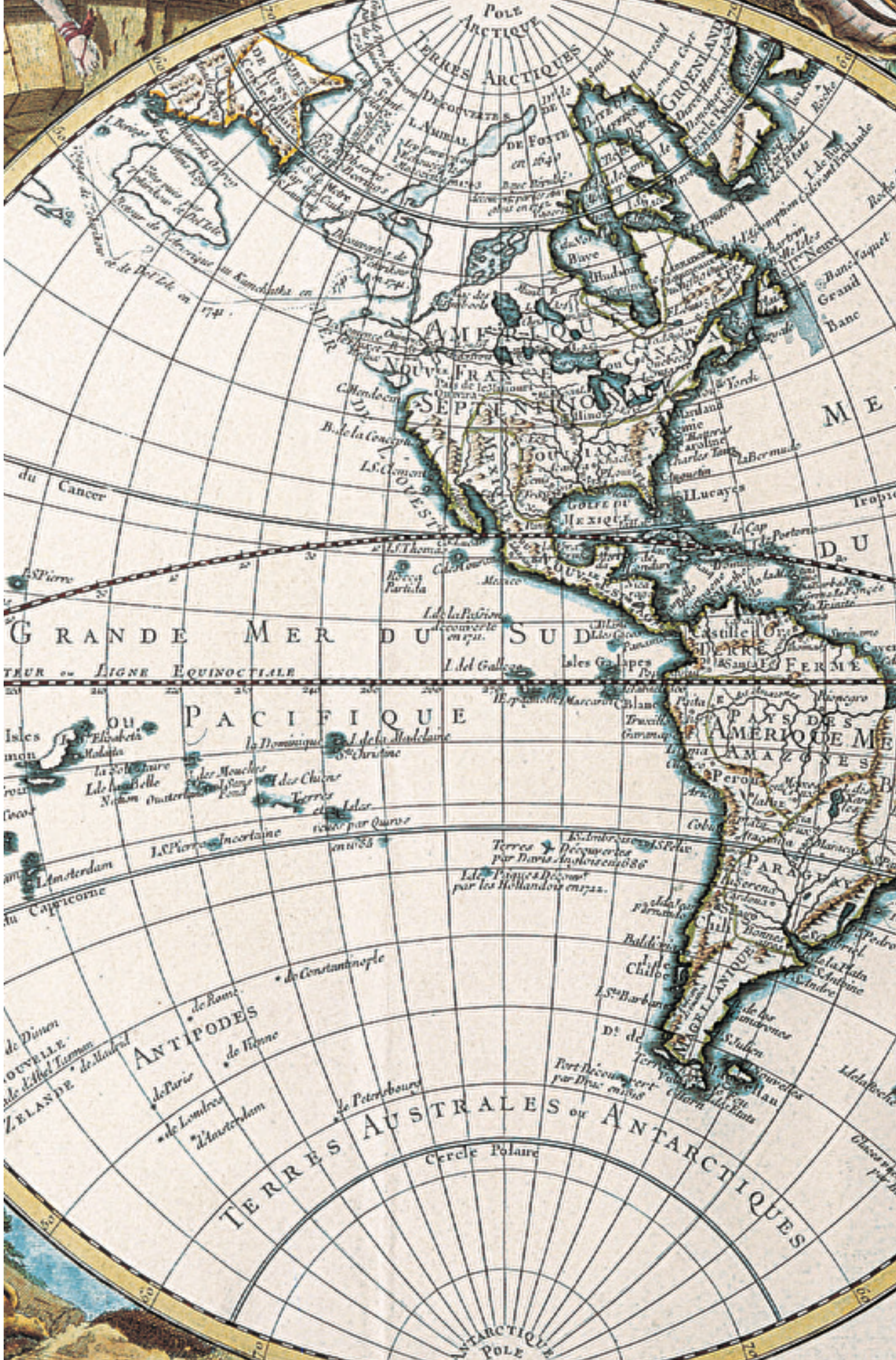
LINE SEGMENT

```
(define (line-segment a b))
(define (segment-start s))
(define (segment-end s))
```

In this representation, a and b are cartesian points

NOTE OMISSIONS

- What have I left out?
  - How a point is constructed using pair notation!
  - How point element access is constructed using pair notation!
  - How a square root is computed!
- This means I could conceivably write programs using these Cartesian descriptions, without knowing how they are implemented!



HOMEWORK 6

HOMEWORK 6: EXTENDING CARTESIAN SYSTEM

- In Scheme:
- create a definition of a rectangle
  - (one possible construction: a pair of pairs of points for corners)
  - (one possible construction: an origin and a size)
  - Assume all rectangles are right-oriented! (no angled rectangles)
- Write a procedure to compute the perimeter of a rectangle
- Write a procedure to compute the area of a rectangle
- Write a procedure to get each of the four corners of a rectangle

- Lastly, for fun: what is your favorite use of a computer *other* than programming?