ITERATORS

Also, review for the exam

> StackOverflow is only useful for people who already understand the answer, but don't have time to figure out exact syntax.
>
> *-Rick Burgess*

ITERATORS
FILTER
MAP
FOLD

ITERATORS

- An iterator allows you to perform a task on a sequence of items in turn
- The iterator is responsible for starting the list at the beginning, processing each item sequentially, and ending when there are no more items
- In Rust, iterators are *lazy*; they have no effect until the first function which consumes that iterator is called

CONSUMING ITERATORS

- When an iterator is used, it is said to be *consumed*.
- Iterators cannot be reused, they contain internal state

ITERATOR SAMPLE

```
fn main() {
    let v = vec![0,1,2,3];

    let mut iter = v.iter();

    println!("{:?}", iter.next());
    println!("{:?}", iter.next());
    println!("{:?}", iter.next());
    println!("{:?}", iter.next());
    println!("{:?}", iter.next());
}
```

Some(0)

Some(1)

Some(2)

Some(3)

None

NEXT() RETURNS AN OPTION

- Please note a couple of things about *next*:
  - Next returns an *option* to a value. This shouldn't be a surprise, since we're working with a Vec<T>.
  - Next returns *None* if we go past the end of the Vec<T>
  - The iterator must be *mutable* in order to modify it via *next*

TRANSFORMING ITERATORS

- An iterator can easily be changed from one type into another type, or into an iterator with another set of values
- For this we can use *map* and *filter*

MAPPING A VEC<T>

```
fn main() {
    let v = vec![0,1,2,3];

    let v2 = v.iter().map(|x| x+1);

    println!("{:?}", v);
    println!("{:?}", v2);
}
```

```
[0, 1, 2, 3]
Map { iter: Iter([0, 1, 2, 3]) }
```

CONVERTING ITERATORS

- As nice as it is to see a conversion like that, did you notice that the values *did not change*?
- This is because the *map* operation of an iterator is *lazy*, and not processed until it's actually used. Debug printing out does not count as being used.
- We can enforce that use by using *collect()*, which converts an iterator back into a Vec<T>.

COLLECT() CALLS

```
fn main() {
    let v = vec![0,1,2,3];

    let v2: Vec<i32> = v.iter().map(|x| x+1).collect();

    println!("{:?}", v);
    println!("{:?}", v2);
}
```

```
[0, 1, 2, 3]
[1, 2, 3, 4]
```

FILTERING OUT ITEMS

- The *filter* higher order function takes a closure that returns a boolean value.
- Mini quiz: what is the name of an expression that returns a boolean value?
- A new iterator is returned from *filter* that contains only the items which pass through said filter

FILTER SAMPLE

```
fn main() {
    let v = vec![0,1,2,3,4,5,6,7,8,9];

    let i1 = v.into_iter().filter(|x| *x % 2 == 0 );
    let v2: Vec<i32> = i1.collect();

    println!("{:?}", v2);
}
```

```
[0, 2, 4, 6, 8]
```

INTO_ITER

- Instead of just using *.iter()*, we used *.into_iter()*. This changes ownership of the vector, and we therefore are no longer working with references
- Therefore, after using *into_iter*, the original vector is no longer available for our use.

THE ACTUAL FILTER

- The filter is a closure containing a name for the element and a predicate
- Only items passing the predicate will be allowed into the new vector
- We still have to *collect()* the filtered items in order to make the filter actually pass

SUMMING VALUES

- One quick and easy way to use a collection is to call sum on it
- let total: i32 = v.iter().sum();
- I have no intention of using this function, but if I don't mention it, someone in the class will wonder why

FOLDING VALUES

- Rust allows us to *fold* values of an iterator, which performs some sort of accumulation based on an iterator.
- The syntax for the closure is slightly more complex, consisting of an initial value, two variables, and a factor that will be applied to each value
- Fold returns an actual value, not an iterator, and is therefore not *lazy*

FOLD SAMPLE

```
fn main() {
    let v = vec![0,1,2,3,4,5,6,7,8,9];

    let total: i32 = v.iter()
                      .fold(0, |total, x| total + x);

    println!("{:?}", total);
}
```

CHAINING ITERATORS

```
fn main() {
    let v = vec![0,1,2,3,4,5,6,7,8,9];

    let total: i32 = v.into_iter()
        .filter(|x| x % 2 == 0)
        .map(|x| x * 2)
        .fold(0, |total, x| total + x);

    println!("{:?}", total);
}
```

NOT TRANSITIVE

- Iterator chaining is not transitive; please make sure you pay close attention to the order of operations
- Also, if you are going to filter(), the earlier you can do so the more efficient your code will be

MORE FEATURES FOR OUR INTERPRETER

LET'S REWRITE SOME THINGS

- Now that we have map / filter / fold available, let's rewrite our interpreter to take advantage of those methods
- We'll focus on *fold*, as that'll allow us to compute our sums and products very easily

EVALUATE: ADDITION

```
fn evaluate(array: Vec<Primitive>) → i32 {
    let element = &array[0];
    let iter = array.iter();
    match element {
        Primitive::Add ⇒ {iter
.fold(0, |total, next| total + eval_prim(next))},
        _ ⇒ 0
    }
}
```

EVAL_PRIM

```
fn eval_prim(primitive: &Primitive) → i32 {
    match primitive {
        Primitive::Number(val) ⇒ *val,
        Primitive::Add ⇒ 0,
    }
}
```

HOW ABOUT MULTIPLY?

MULTIPLY

```
fn evaluate(array: Vec<Primitive>) → i32 {
    let element = &array[0];
    let iter = array.iter();
    match element {
        Primitive::Add ⇒ {iter.fold(0, |total, next|
total + eval_prim(next))},
        Primitive::Multiply ⇒ {iter.fold(1, |total,
next| total * eval_prim(next))},
        Primitive::Number(val) ⇒ *val
    }
}
```

MULTIPLY

```
fn eval_prim(primitive: &Primitive) → i32 {
    match primitive {
        Primitive::Number(val) ⇒ *val,
        Primitive::Add ⇒ 0,
        Primitive::Multiply ⇒ 1,
    }
}
```

LET'S CHECK IT OUT

(+ 3 4 5) , (* 3 4 5)

```
fn main() {
    let mut primitives = Vec::new();
    primitives.push(Primitive::Add);
    primitives.push(Primitive::Number(3));
    primitives.push(Primitive::Number(4));
    primitives.push(Primitive::Number(5));
    let result = evaluate(primitives);
    println!("result was {}", result);
    let mut primitives = Vec::new();
    primitives.push(Primitive::Multiply);
    primitives.push(Primitive::Number(3));
    primitives.push(Primitive::Number(4));
    primitives.push(Primitive::Number(5));
    let result = evaluate(primitives);
    println!("result was {}", result);
}
```

result was 12

result was 60

CAN WE DO BETTER?

COPY / CLONE A PRIMITIVE

```
#[derive(Copy,Clone)]
enum Primitive {
    Add,
    Multiply,
    Number(i32)
}
```

GET RID OF EVAL_PRIMITIVE

```
fn evaluate(array: Vec<Primitive>) → i32 {
    let element = &array[0];
    let mut iter = array.iter();
    iter.next();
    match element {
        Primitive::Add ⇒ {iter.fold(0, |total, next|
            total + evaluate(vec![*next]))},
        Primitive::Multiply ⇒ {iter.fold(1, |total, next|
            total * evaluate(vec![*next]))},
        Primitive::Number(val) ⇒ *val
    }
}
```

WHAT CHANGED?

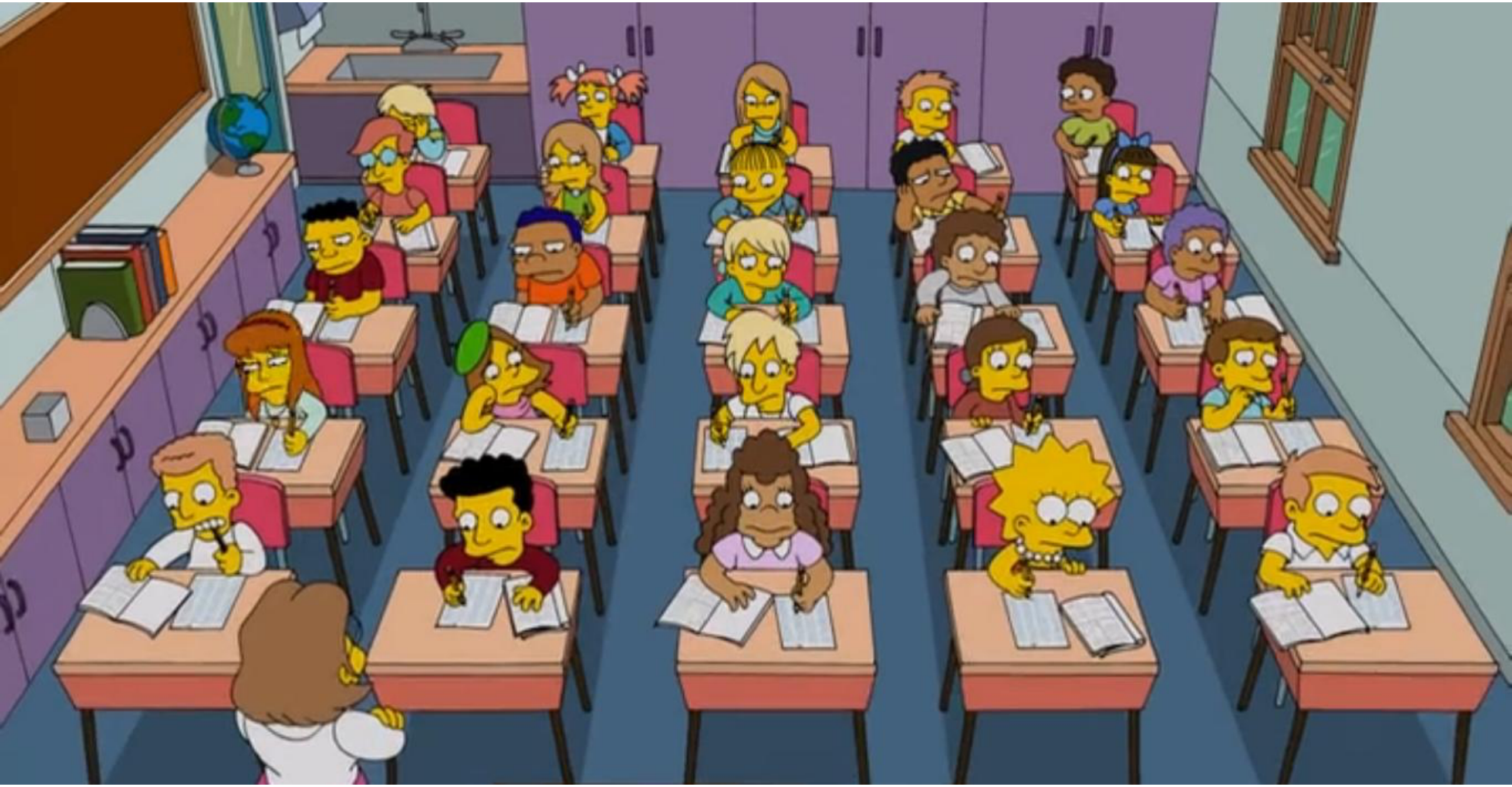
- We get our first operator, and then advance the iterator
- Then we match that operator, and if it's a number, we just return its value
- If it's add or multiply, our iterator is now aiming at the first number
- We then call the appropriate match arm and its fold operation on all of the remaining numbers

LOOKING AT JUST THE ADD ARM

```
Primitive::Add ⇒ {iter.fold(0, |total, next|
                    total + evaluate(vec![*next]))},
```

WHAT'S BETTER?

- We don't have a redundant eval_prim operator!
- We can add or multiply any quantity of number primitives!
- These will streamline our code, and simplify our lives in the future. I am sure our future selves will thank us for this.



EXAM REVIEW

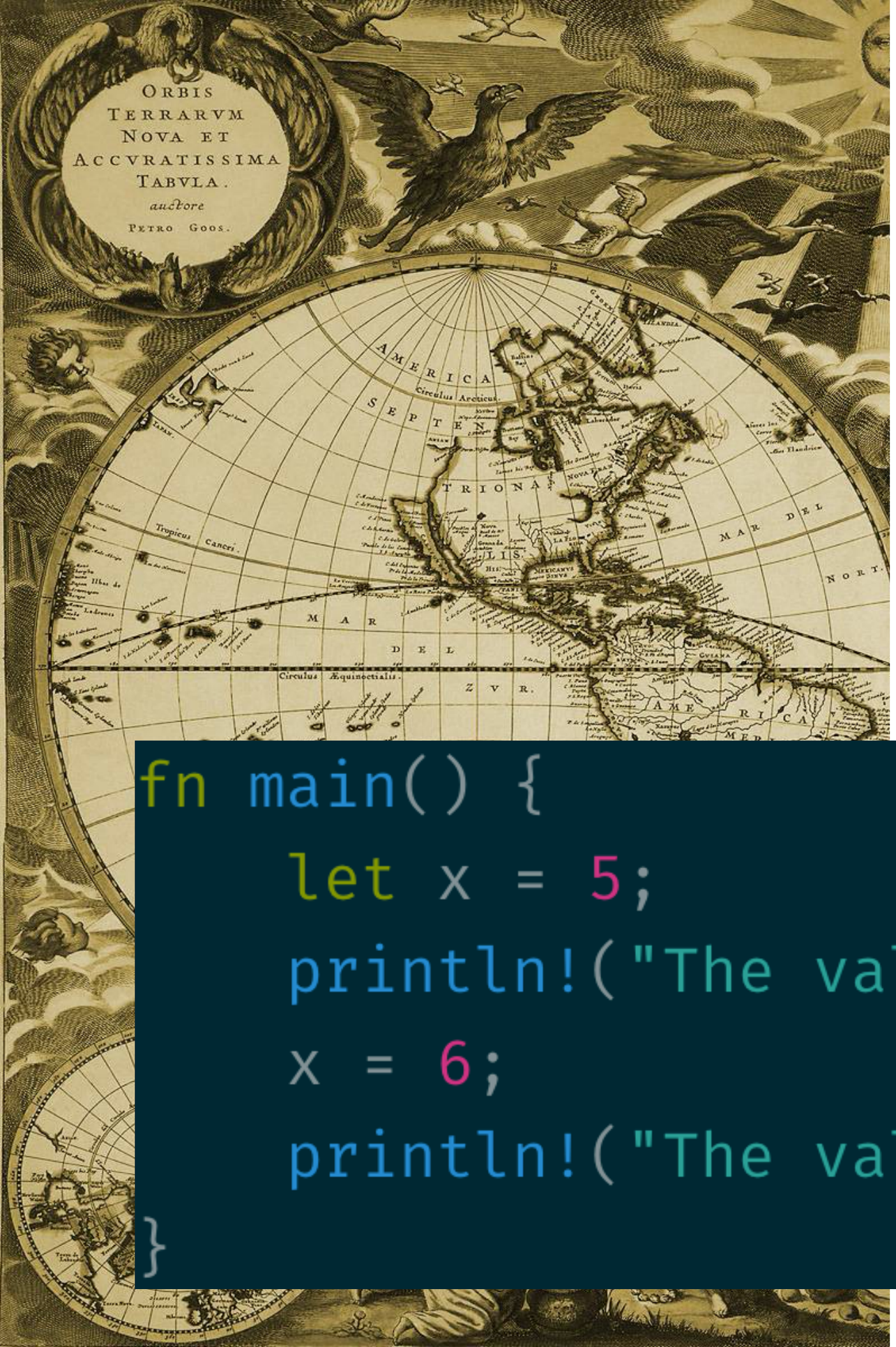

A. LAGOA
S. Lazero
S. Paulo
S. Matheus
S. Thome novo
Terreiro do Vizegy
As cazas do Vizegy
Champanas e cotias de gentios
S. Ioseph
N.S. de Monte
S. Amaro
S. Luzia
A ILHA DE
ILHA DE
NORVA

EXAM FORMAT

- 25 short answer questions
  - 10 must-answer
  - Pick 10 of the remaining 15 to answer
- I do not believe any of the questions are "trick" questions. I don't have time for that, I'm not interested in proving my smartness at your expense.
- No questions over homework
- **Please** use a pen

CARGO

- Rust's package manager is *cargo*
- Cargo handles:
  - Dependencies
  - Building
  - Unit tests
  - Project setup



DEFAULT BEHAVIOR

- By default, variables in Rust are immutable
- Adds safety and easy concurrency
- Guarded at compile time

```
fn main() {
    let x = 5;
    println!("The value of x is {}", x);
    x = 6;
    println!("The value of x is {}", x);
}
```

MUTABILITY

- We generally prefer immutable variables; this will make our code more robust and make it more difficult to end in unexpected changes.
- Mutability is possible with the `mut` keyword.
- This gives information to the compiler and also to future programmers
- Mutability may be preferred if it can prevent a large memory allocation, such as with large structures

CONSTANTS VS VARIABLES

- Although *let* may create an *immutable* variable, it is nonetheless a variable.
- Rust also has *constants* available.
- Constants must have types annotated. We'll get to that next.
- Constants may not be set to the result of a function call, they may only be set with a truly constant value

```
const UPPER_LIMIT: u32 = 128_000;
```

SHADOWING VARIABLES

- You can declare a new variable with the same name as a previous variable
- The new variable *shadows* the previous variable
- This is not the same as using a *mut* variable
- We can even change the type of a value as we reuse it!

SCALAR TYPES

- Scalar types represent a *single* value
- There are four primary categories of scalars in Rust:
  - Integers
  - Floating Point Numbers
  - Booleans
  - Characters

TUPLES

- Tuples are a general method for grouping some number of values with a variety of types into one compound type
- Tuples have a fixed length
- Tuples are declared by creating a set of comma-separated values within parenthesis
- Each position in the tuple has a fixed type
- Each position in the tuple may have a different type
- We can use type annotation or inference, just as with scalar variables

ARRAYS

- Arrays are collections of values with the *same* type
- Arrays have a *fixed length* as well as a single type
- Arrays are declared with values in a comma-separated list inside of square brackets
- Arrays aren't as flexible as some other data types (such as vectors) but are usually faster — they are allocated on the stack instead of the heap

EXPRESSIONS

- One type of procedure we will be working with extensively is an *expression*
- Expressions can be considered a combination of one or more constants, variables, operators, and functions that the programming language interprets and computes to produce another value

PRIMITIVE PROCEDURES

- Expressions that represent numbers can be combined with expressions representing a primitive procedure to form a compound expression
  - (+ 5 5)
  - (- 17 3)
  - (* 2 2)
  - (+ 2.2 2.3)
- These are called *combinations* when a list of expressions are contained within parenthesis

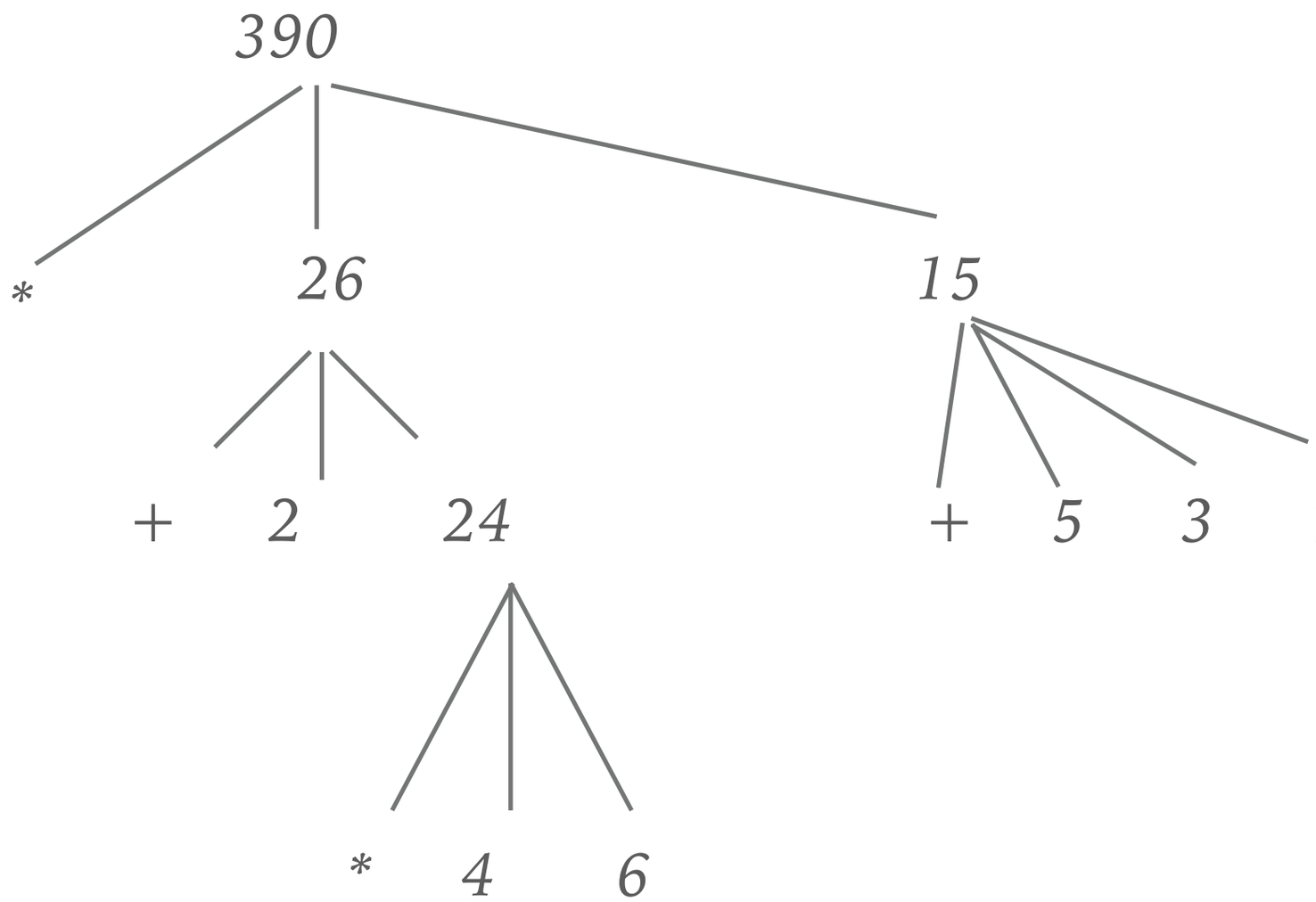
PREFIX NOTATION

- Scheme prefers *prefix notation*
- Prefix notation looks odd from a mathematical standpoint
- Prefix notation makes it much simpler to accommodate procedures with an arbitrary number of elements
  - (+ 1 2 3 4)
  - (* 2 2 2)

EVALUATING COMBINATIONS

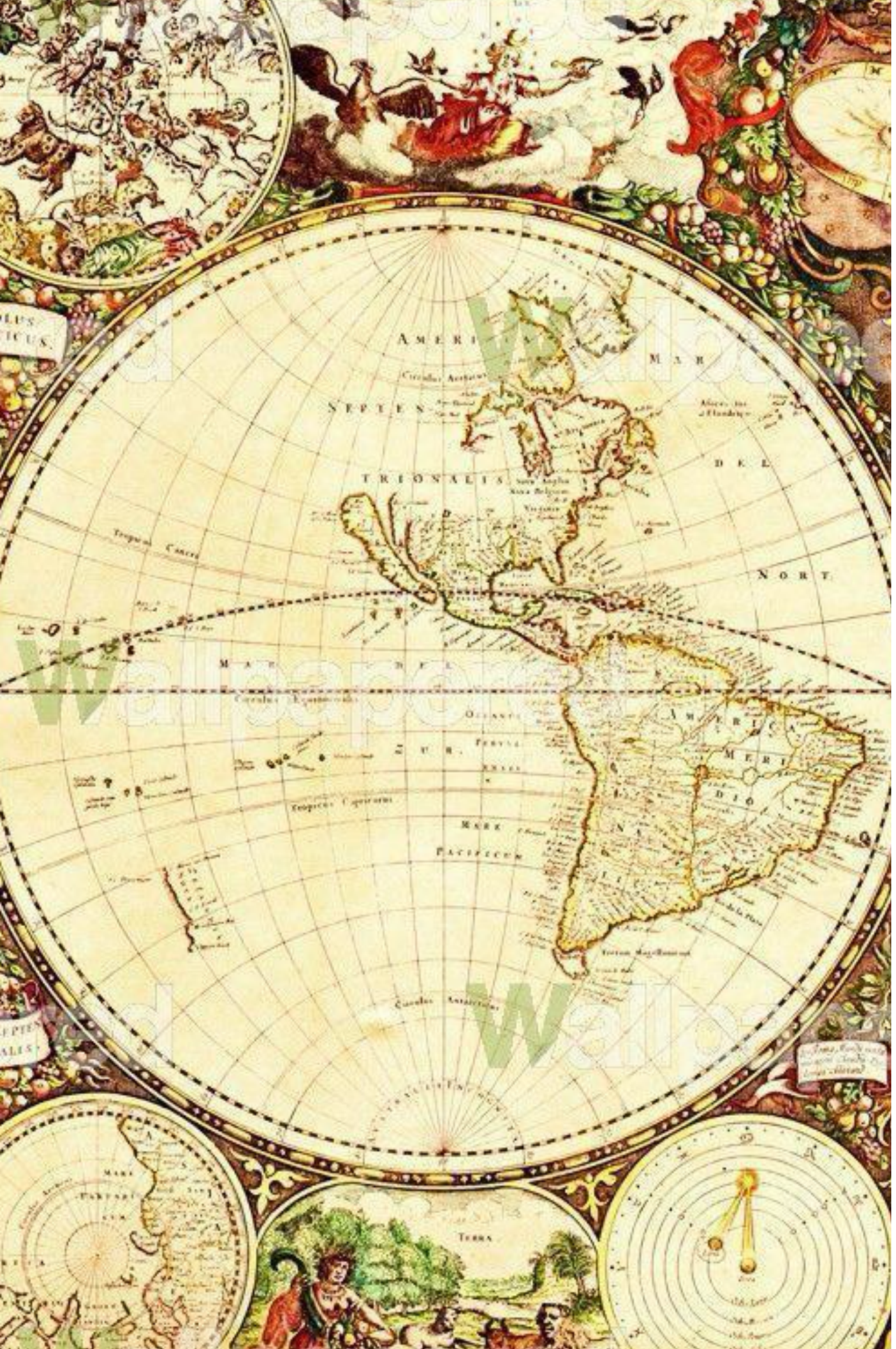
- Two-step process
  - Evaluate the subexpressions of the combination
  - Apply the procedure that is the value of the leftmost subexpression (the *operator*) to the values of the other subexpressions (the *operands*)
- This is a recursive process, because the first step says "do this same thing to the subexpressions

(* (+ 2 (* 4 6)) (+ 5 3 7))



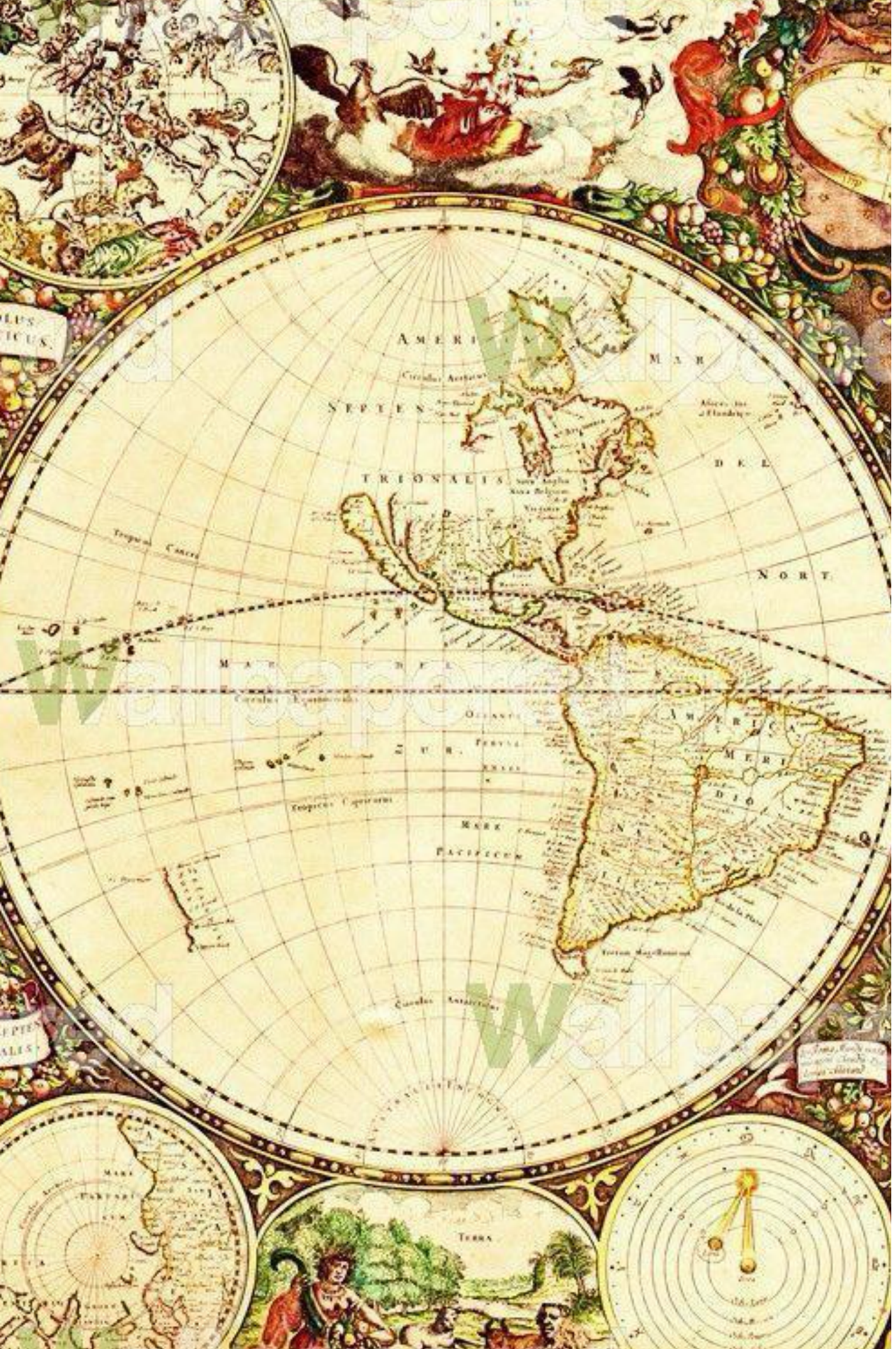
COMPOUND PROCEDURES

- In scheme, we can define a procedure with a name, and thereafter refer to it as a unit
- We use the *define* keyword, just as we might for a variable
- (define (square x) (* x x))
- The general form is:
- (define (<name> <formal parameters>) <body>)



EVALUATION OF COMBINATIONS

- We use the same technique as we use for primitive procedures:
  - Evaluate the elements of the combination
  - Apply the procedure (operator) to the arguments (operands)
- *To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument*

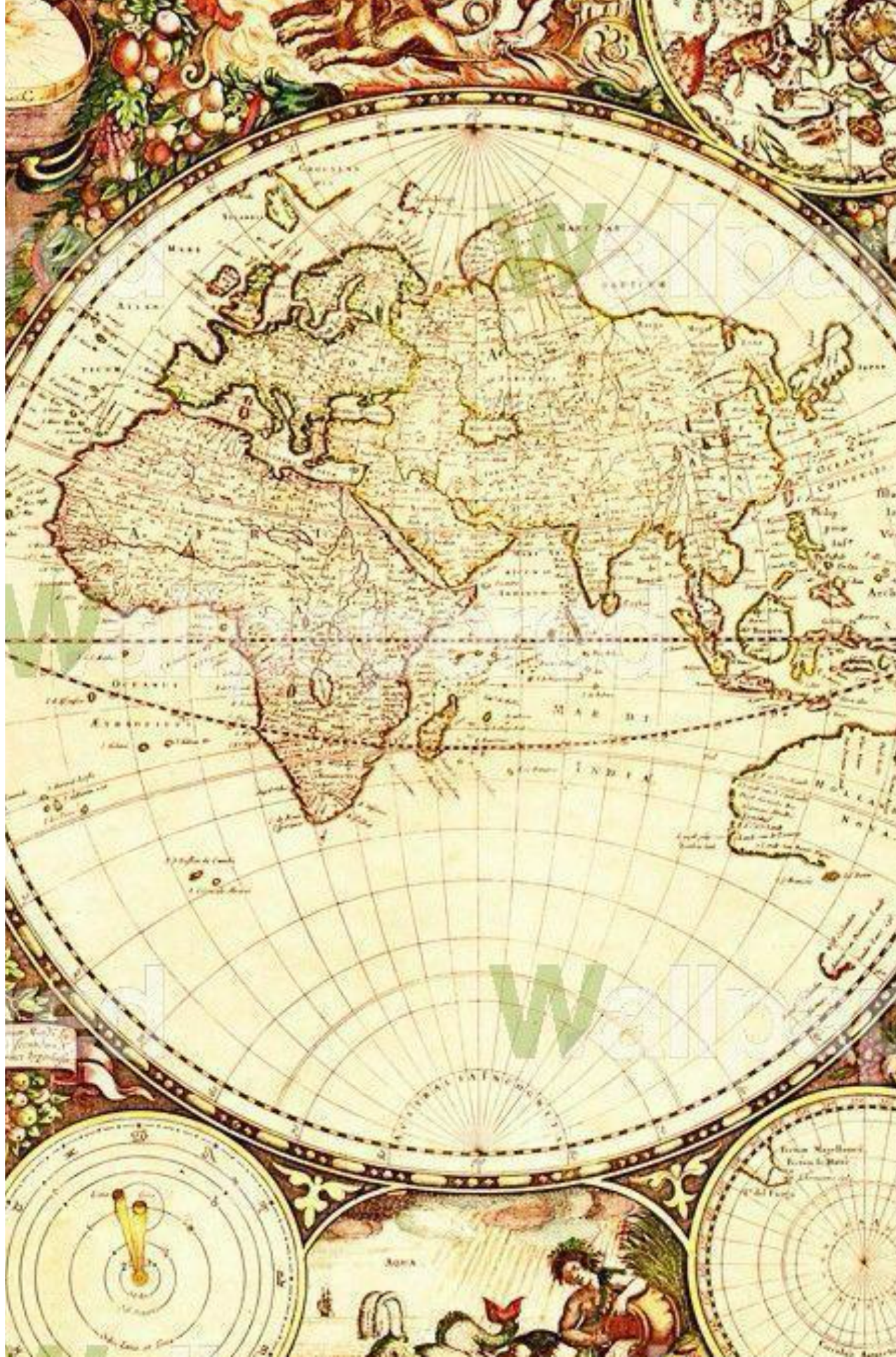


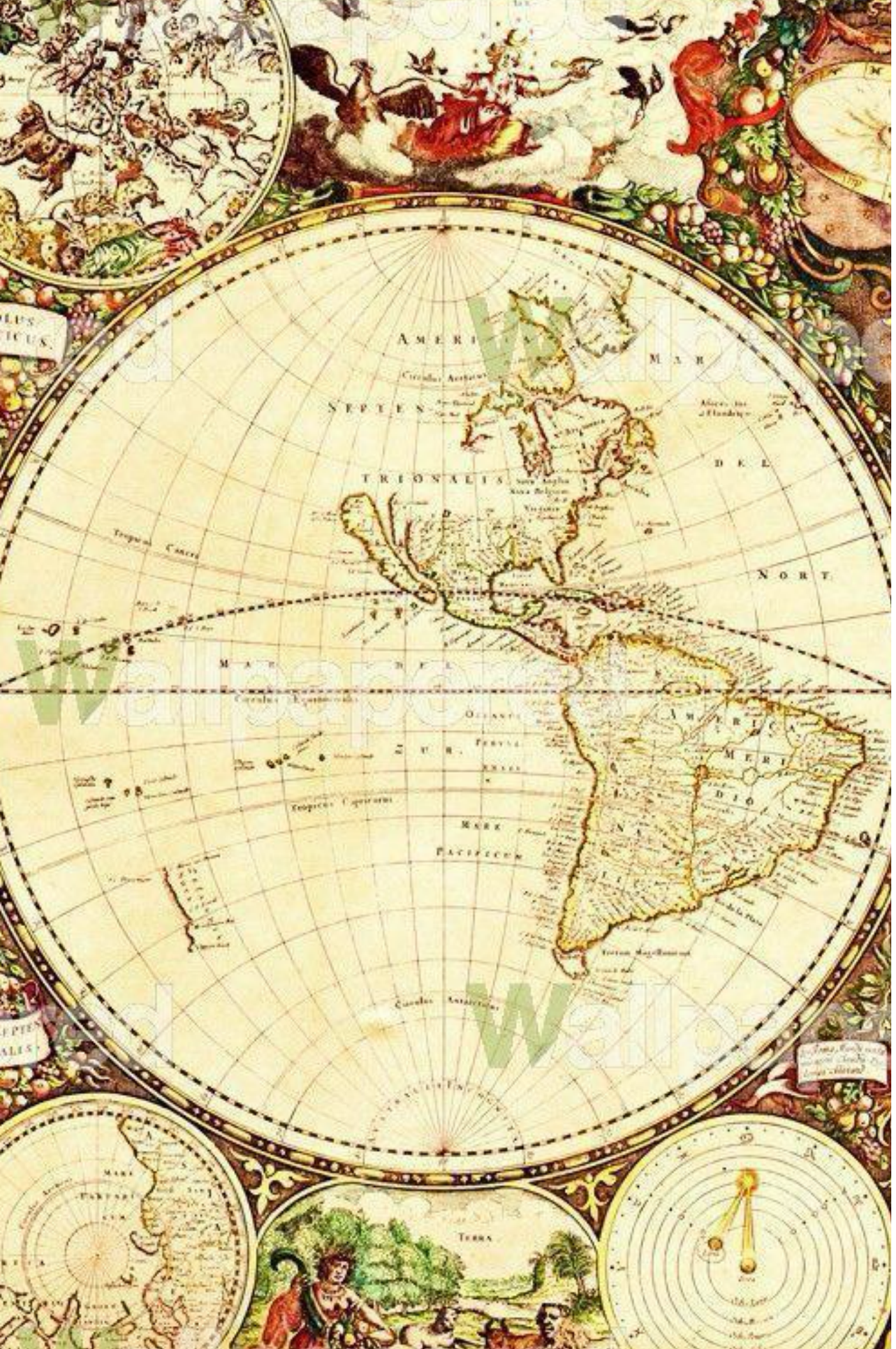
SUBSTITUTION MODEL

- Replacing expressions in this way is called the substitution model for procedure application
- This helps us think about procedure application, but doesn't (necessarily) reflect how an interpreter really works
- Interpreters rarely perform textual manipulation, for example

NORMAL ORDER

- In normal order, we fully expand and then reduce.
- Generally will produce same results as applicative order
- Normal order isn't typically used





CONDITIONAL EXPRESSIONS

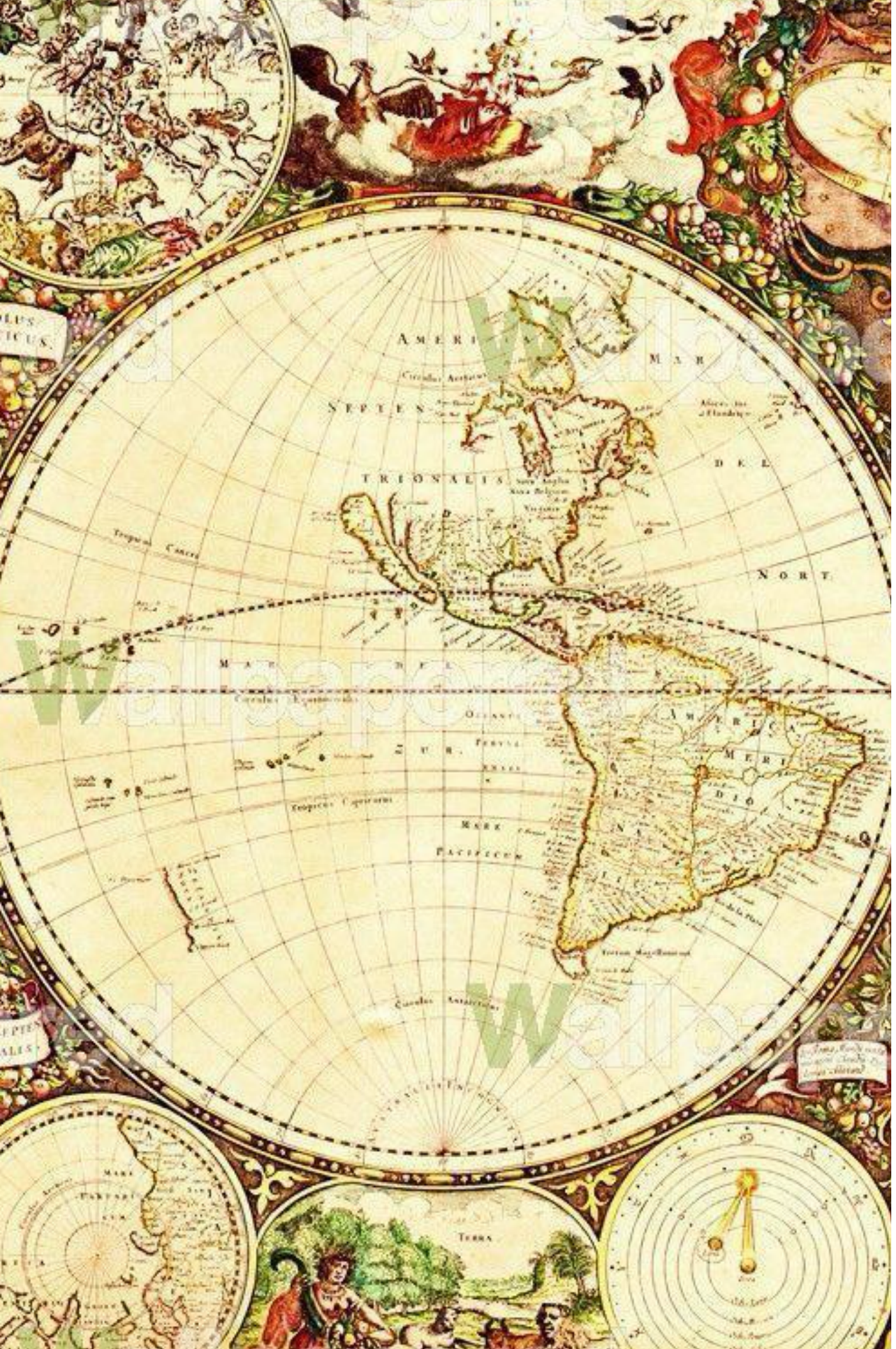
- In order to have branching available to our programs, we must provide some kind of conditional expression in our language
- Conditional expressions can also be called *case analysis*, notating expressions which are evaluated

GENERAL FORM OF CONDITIONAL EXPRESSIONS

```
(cond (<p1> <e1>)
      (<p2> <e2>)
      . . .
      (<pn> <en>))
```

THE NAMES OF THINGS

- Each parenthesized pair following *cond* is called a *clause*
- The first expression in each pair is a *predicate*
  - A predicate is an expression whose value is interpreted as either true or false
- The second expression in each pair is a *consequent expression*

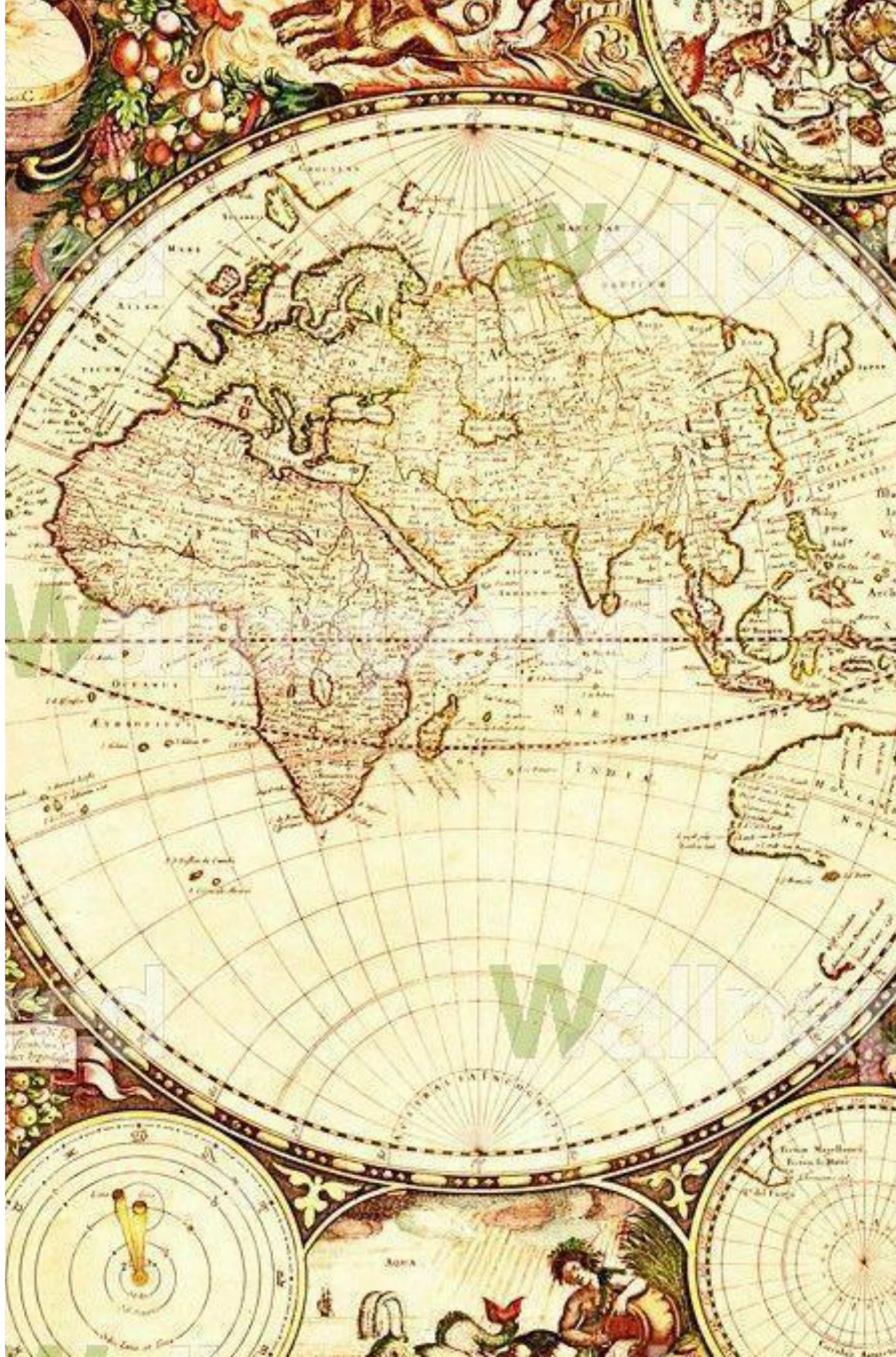


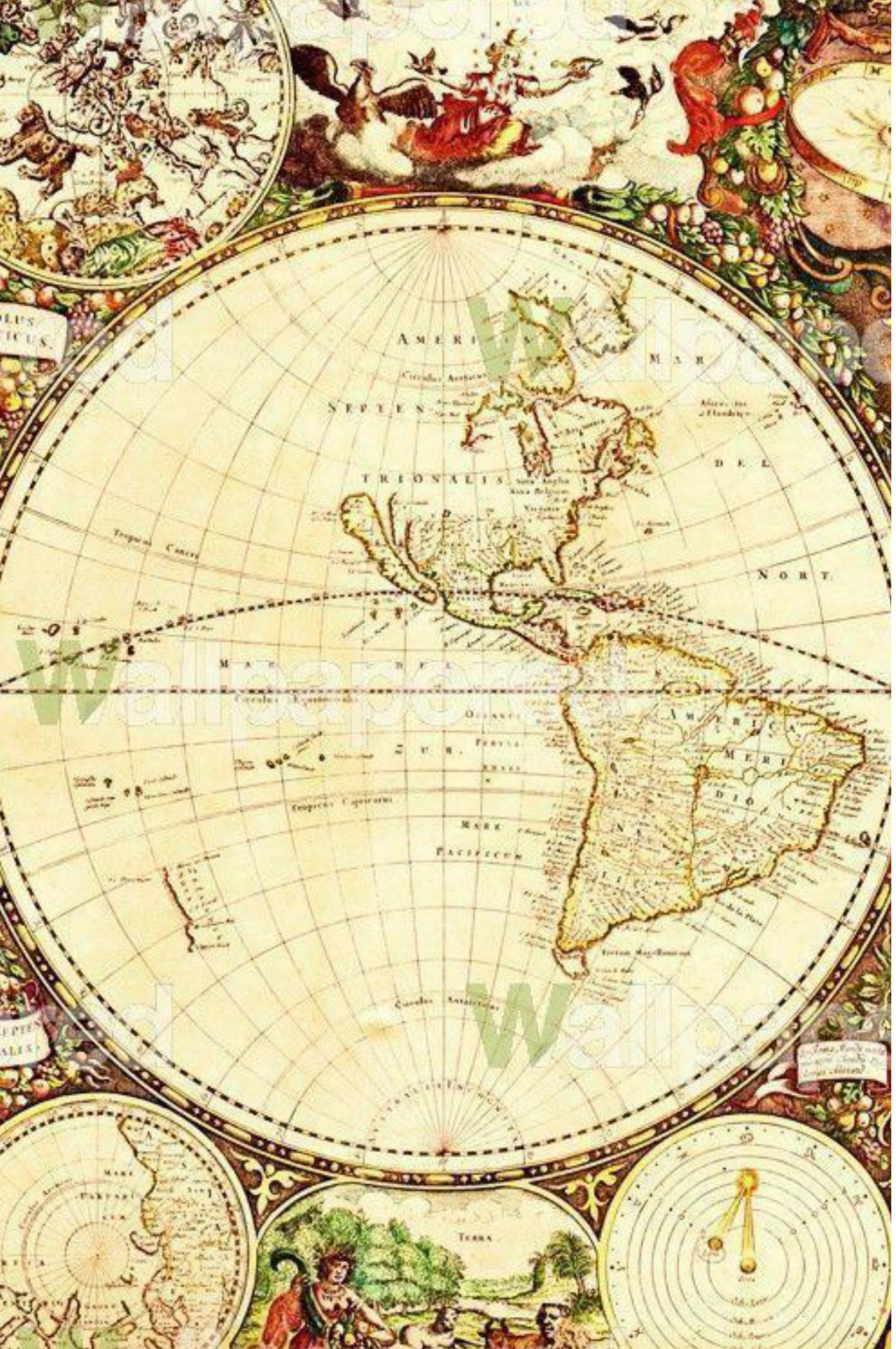
FUNCTION CONVENTIONS

- Functions are conventionally named using snake case
- Functions start with *fn* and have a set of parenthesis after the function name
- Curly braces indicate the beginning and end of the function body

FUNCTION PARAMETERS

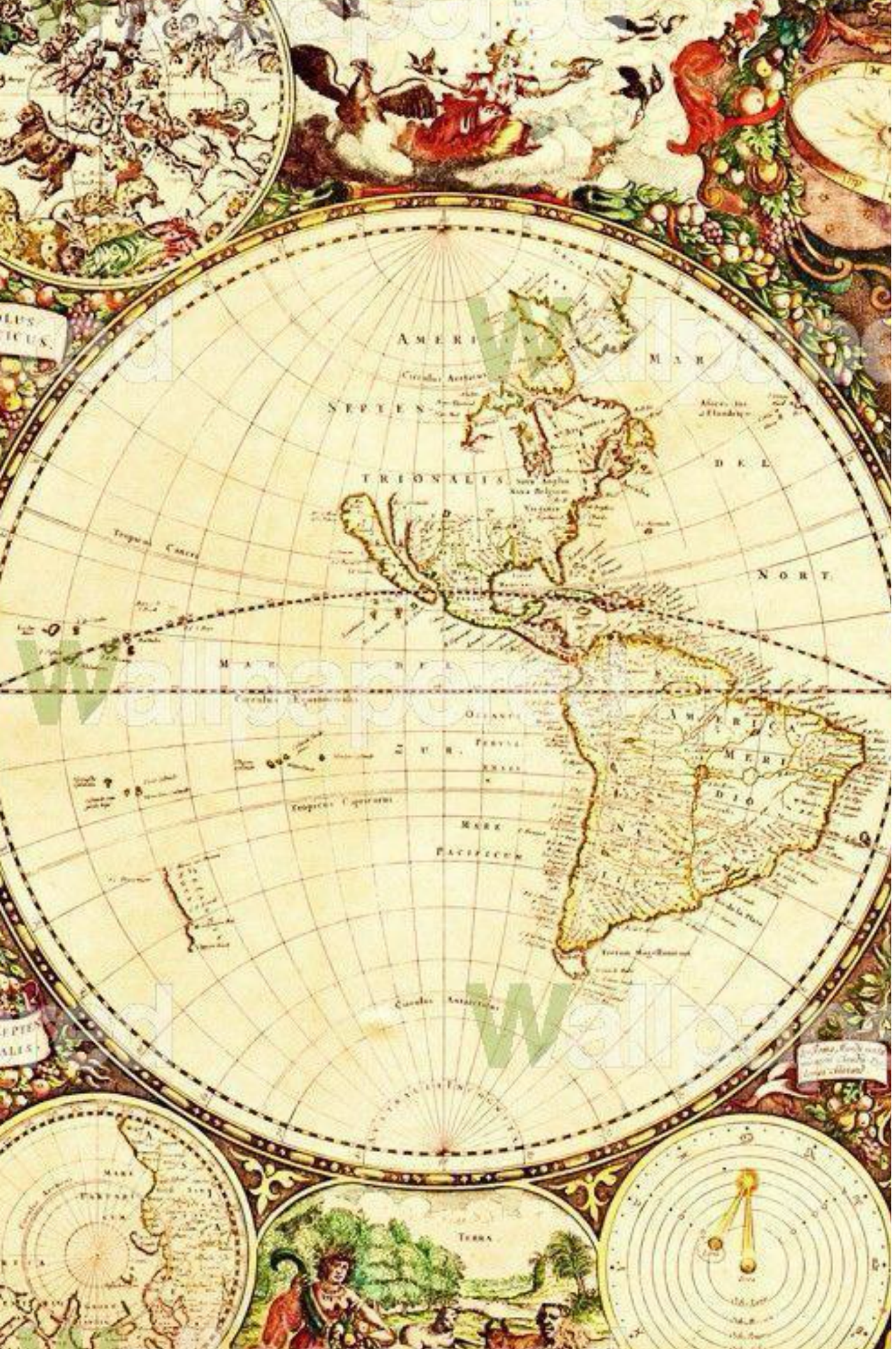
- Functions can be defined to have *parameters*, which are variables that are part of a function's *signature*
- The values themselves are called *arguments*
  - The distinction between a parameter and an argument is academic. It won't be on the test. Call it either one, I don't care
- Parameters *must* have defined types in the function signature





STATEMENTS & EXPRESSIONS

- The body of a function contains *statements* and *expressions*
- A *statement* is an instruction that performs an action and does not return a value
- An *expression* evaluates to a resulting value



RETURN VALUES

- Each function may return a value to the code which calls them
- Return values must have their type defined in the function signature, preceded by an arrow (->)
- A function may return early by using the *return* keyword with a value, or the final expression of a function will be returned implicitly

APPLICATIVE VS NORMAL

- Under applicative order, all arguments are evaluated when a procedure is applied
- Under normal order, no arguments are evaluated until the actual argument values are needed
- Sample procedure:

```
(define (triple x) (* x 3))
(define (half x) (/ x 2))
(define (f x) (half (triple x)))
(f 6)
```

MEMORY MANAGEMENT STYLES

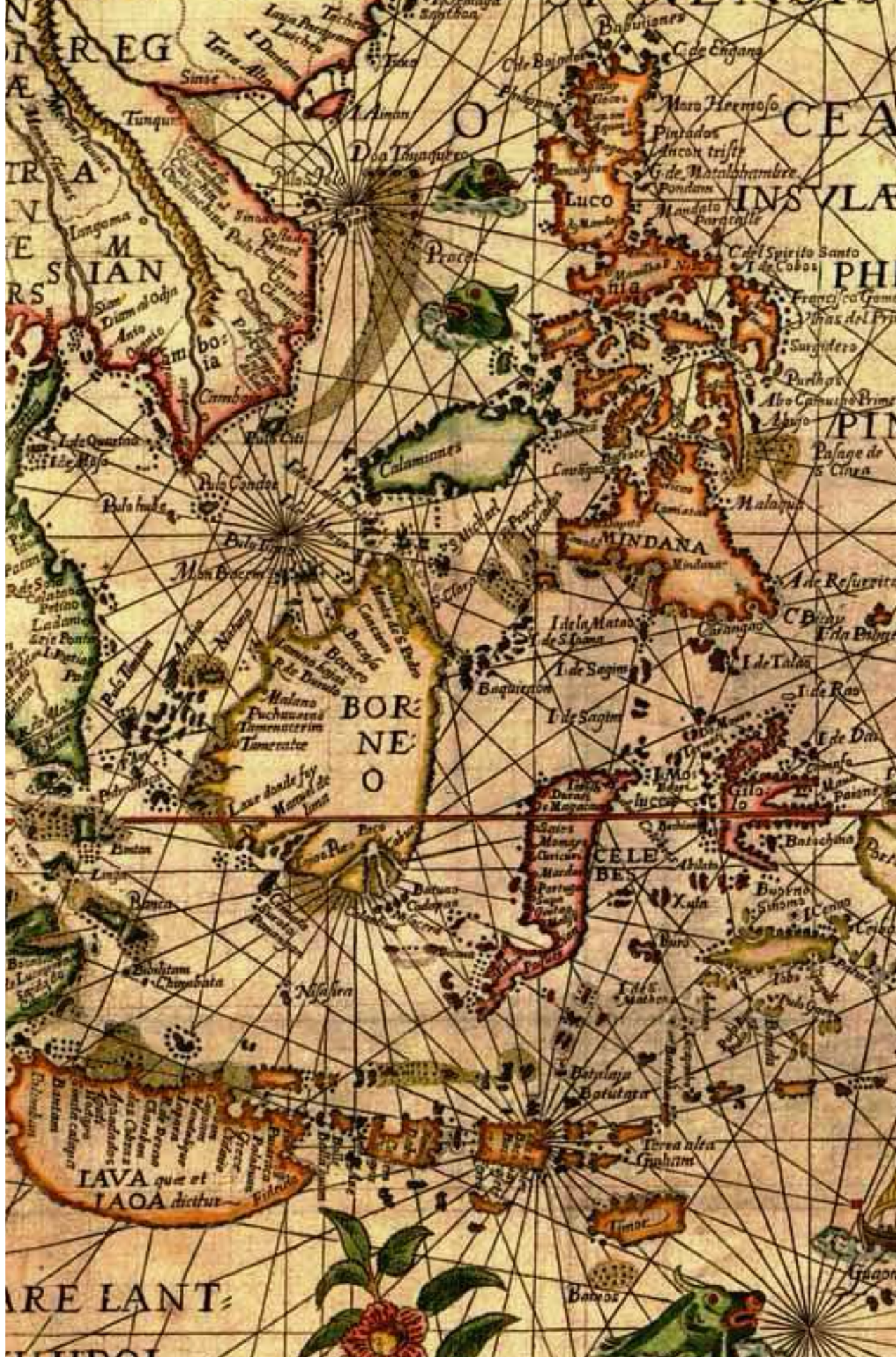
- There are 2 common ways to use a computer's memory while a program is running
  - Garbage collection
  - Explicit allocation and release
- Rust uses a third approach:
  - Compiler works through a set of rules at compiler time
  - No ownership features slow down the program as it runs

RULES OF OWNERSHIP

- Each value in Rust has a variable that is called its *owner*
- There can only be one *owner* for a given value at any time
- When the owner goes *out of scope*, the value will be dropped

BORROWING

- When we borrow a value, we do not have any ownership of that value
- Values are returned to the referring function once the borrowing function is complete
- Borrowing does not allow mutation, of course

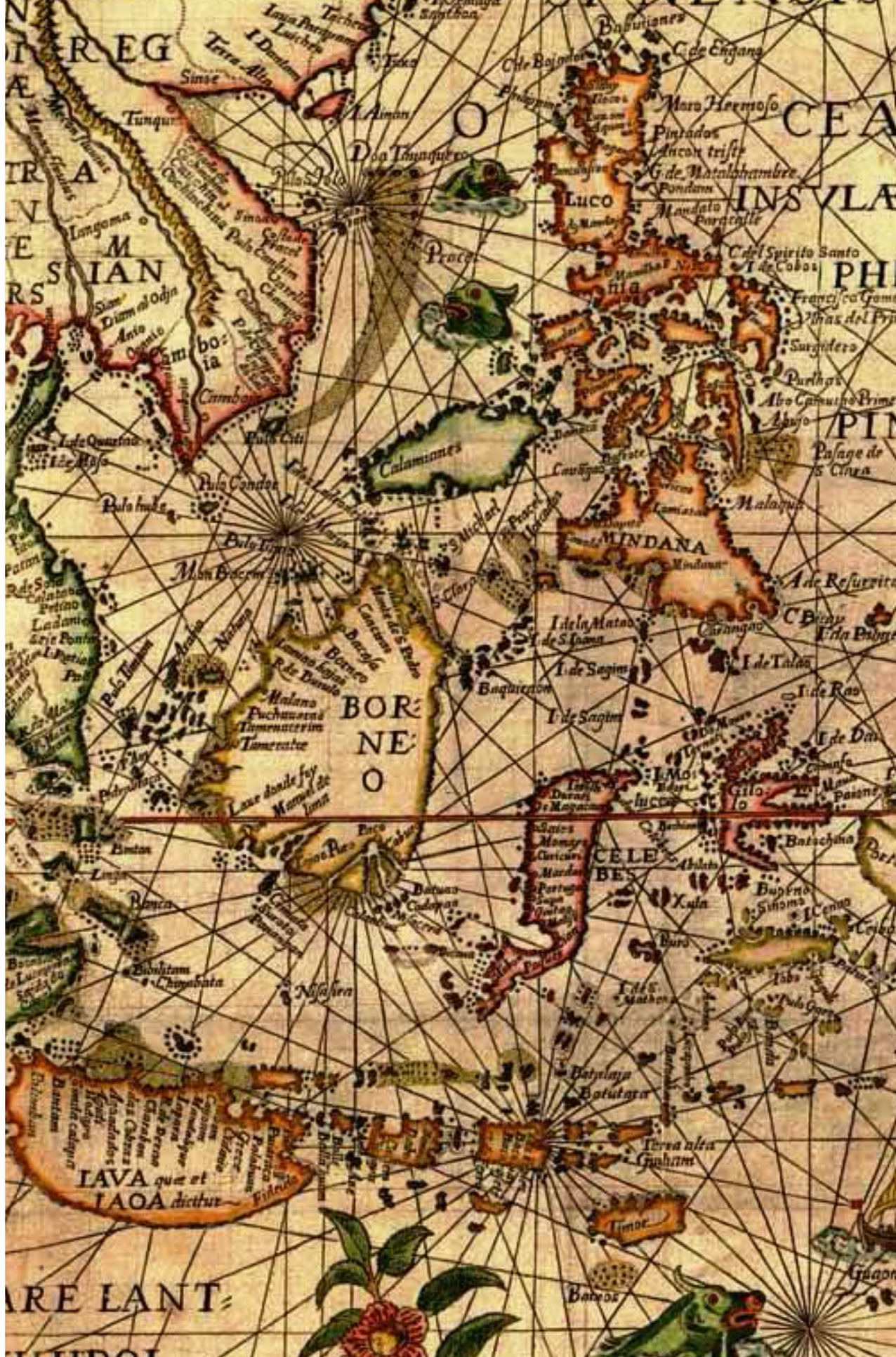


MUTATING REFERENCES

- It should not come as a surprise that you're not allowed to mutate references
- Rust prefers immutable variables everywhere else, and so with references as well

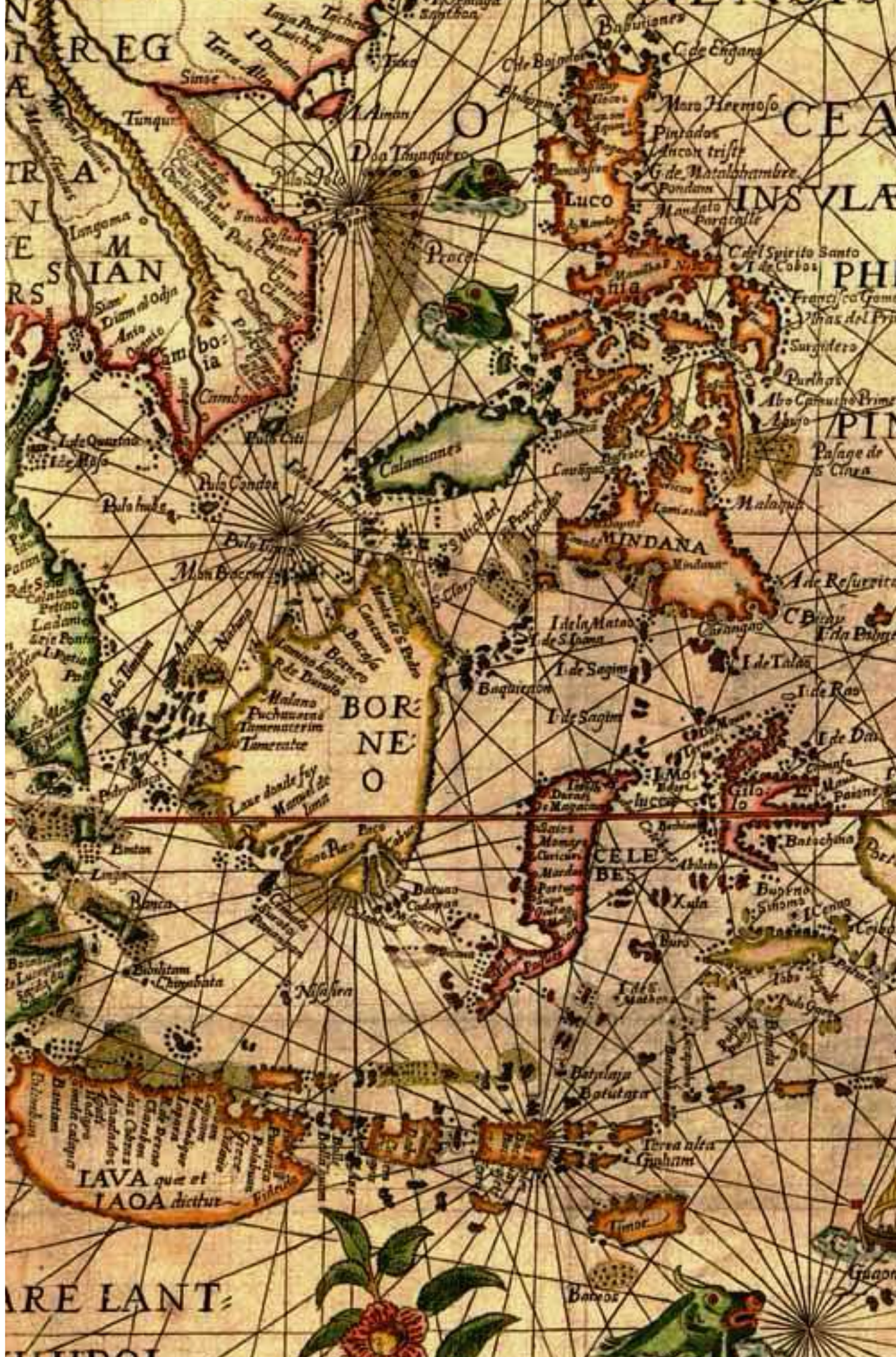
RESTRICTIONS

- There is one *significant* restriction to mutable references!
- You can have only *one* mutable reference to any particular piece of data in any given scope!



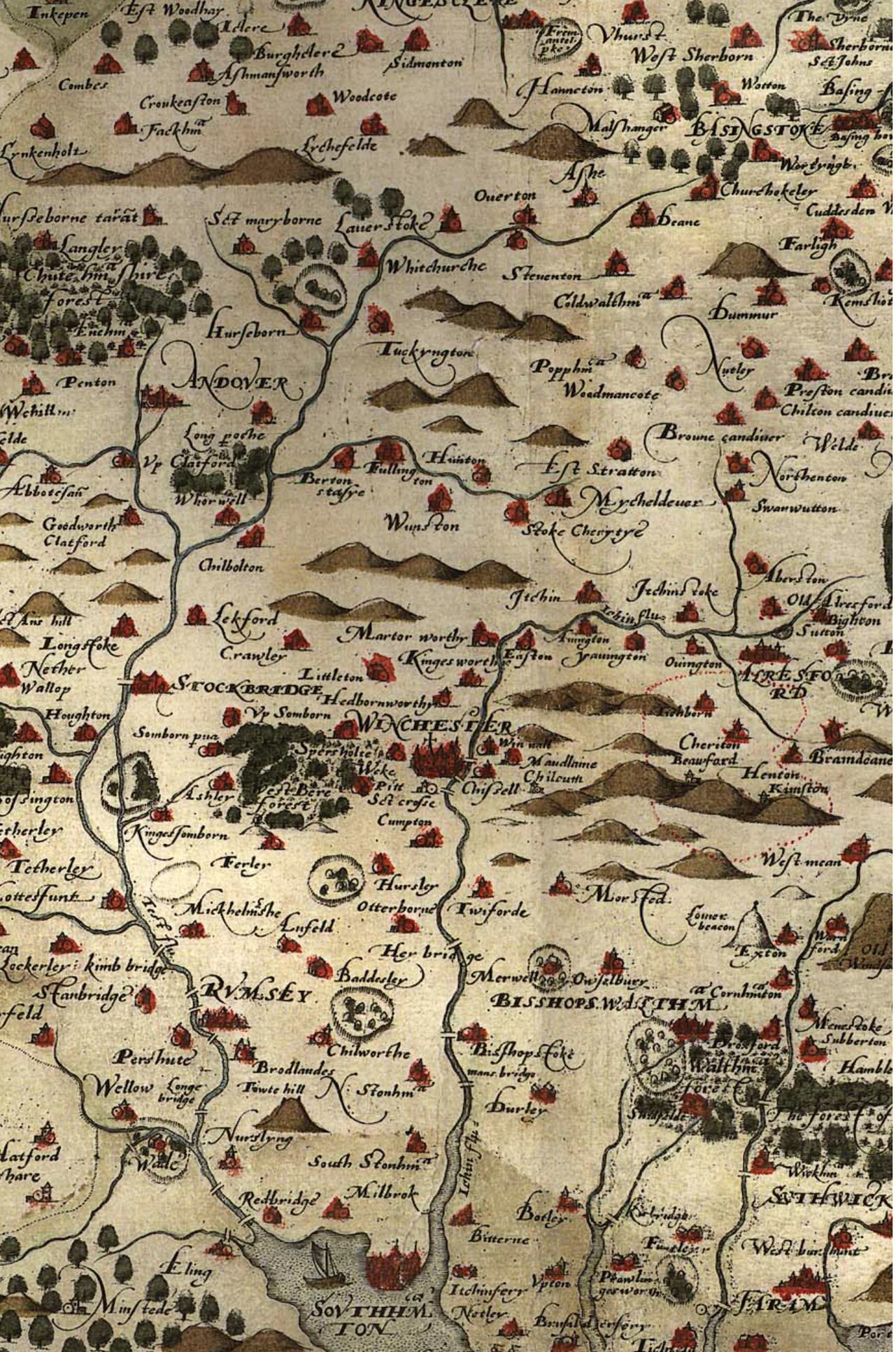
DATA RACES

- Three conditions make a data race:
  - Two or more pointers access the same data at the same time
  - At least one of the pointers is being used to write to that data
  - There is no mechanism being used to synchronize access to the data



DANGLING REFERENCES

- The Rust compiler will also not allow us to have a *dangling reference*, or a reference to memory which has already been dropped
- This is done, as with all other things, at compile time



TWO DIFFERENT SHAPES

- The first implementation of the Peano addition is *linear iteration*
- The second implementation of the Peano addition is *linear recursion*
- In each step of the *linear iteration*, that step has all of the information needed to complete the computation
- In the *linear recursion*, each step depends on state carried from step to step

RECURSION

- Don’t confuse a recursive *process* with a recursive *procedure*
- A recursive *procedure* refers either indirectly or directly to the procedure itself
- A recursive *process* speaks as to how the process evolves, not the syntax of how it is written

TREE RECURSION

- Often results in redundant computation
- Entire subsets of the tree are duplicated
- Grows exponentially

USING LAMBDA

```
(lambda (x) (+ x 4))
(lambda (x) (/ 1.0 (* x (+ x 2))))


(define (pi-sum a b)
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b))
```

LAMBDA IS JUST LIKE DEFINE

- We define lambdas the same way we define procedures
- We just avoid providing a name for the procedure
- (lambda (<formal-params>) (<body>))
- These are *every bit* as much of a procedure as if we had used define, but they are not given a name in the environment

STRUCT DEFINITION

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool
}

fn main() {
   let user = User {
        email: String::from("foobar@dk"),
        username: String::from("Russell"),
        active: true,
        sign_in_count: 10,
    };
    println!("{}",user.username);
}
```

TUPLE STRUCTS

- Tuple structs are the ability to create a new type, based on a tuple, without naming the individual fields
- Useful if we want to know a specific *type* for a tuple, but naming the particular fields are overly verbose or redundant

USING STRUCTS: WHY?

- The use of Structs allows us to group data together, which improves the meaning of our program
- It not only makes the program more clear at the point of *area*, but it also gives more meaning to the simple *width* and *height* parameters
- Don't get me started on the tuple

BUT CHECK OUT THIS ERROR MESSAGE!

```
error[E0277]: `Rectangle` doesn't implement
`std::fmt::Display`
 ⟶ src/main.rs:7:32
  |
7 |     println!("The rect is {}", rect);
  |                                ^^^^ `Rectangle` cannot be
formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented
for `Rectangle`
  = note: in format strings you may be able to use `{:?}` (or
{:#?} for pretty-print) instead
  = note: required by `std::fmt::Display::fmt`
```

STRUCT METHODS

- Struct methods are defined just like any other function with the *fn* keyword. They can have parameters and a return value.
- Methods are only valid within the context of a struct, however.
- The first parameter of any method is *self*, meaning the instance of the struct the method is being called upon.

RECTANGLE AREAS

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32
}
impl Rectangle {
    fn area(&self) → u32 {
        self.width * self.height
    }
}
fn main() {
    let rect = Rectangle { width: 30, height: 50 };
    println!("Rectangle area is {}", rect.area());
}
```

ASSOCIATED FUNCTIONS

- Associated functions are functions within an *impl* block which do not take *self* as a parameter
- These are *functions*, not *methods*, because they do not have an instance of the struct that they work from
- Often used for constructors

ASSOCIATED FUNCTION FOR CONSTRUCTION

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32
}
impl Rectangle {
    fn area(&self) → u32 {
        self.width * self.height
    }
    fn square(size: u32) → Rectangle {
        Rectangle{ width: size, height: size }
    }
}
fn main() {
    let rect = Rectangle::square(30);
    println!("The rect is {:#?}", rect);
    println!("Rectangle area is {}", rect.area());
}
```

ASSOCIATED DATA

- Data can be associated with enumerations as well
- Associated data may be of the same type for each enumerated value, or different for each enumerated value

ATTACHED DATA

```
#[derive(Debug)]
enum UniversityPerson {
    Student(f32),
    Instructor(f32)
}
fn main() {
    let s = UniversityPerson::Student(1.75);
    println!("{:?}", s);
}
```

MATCH

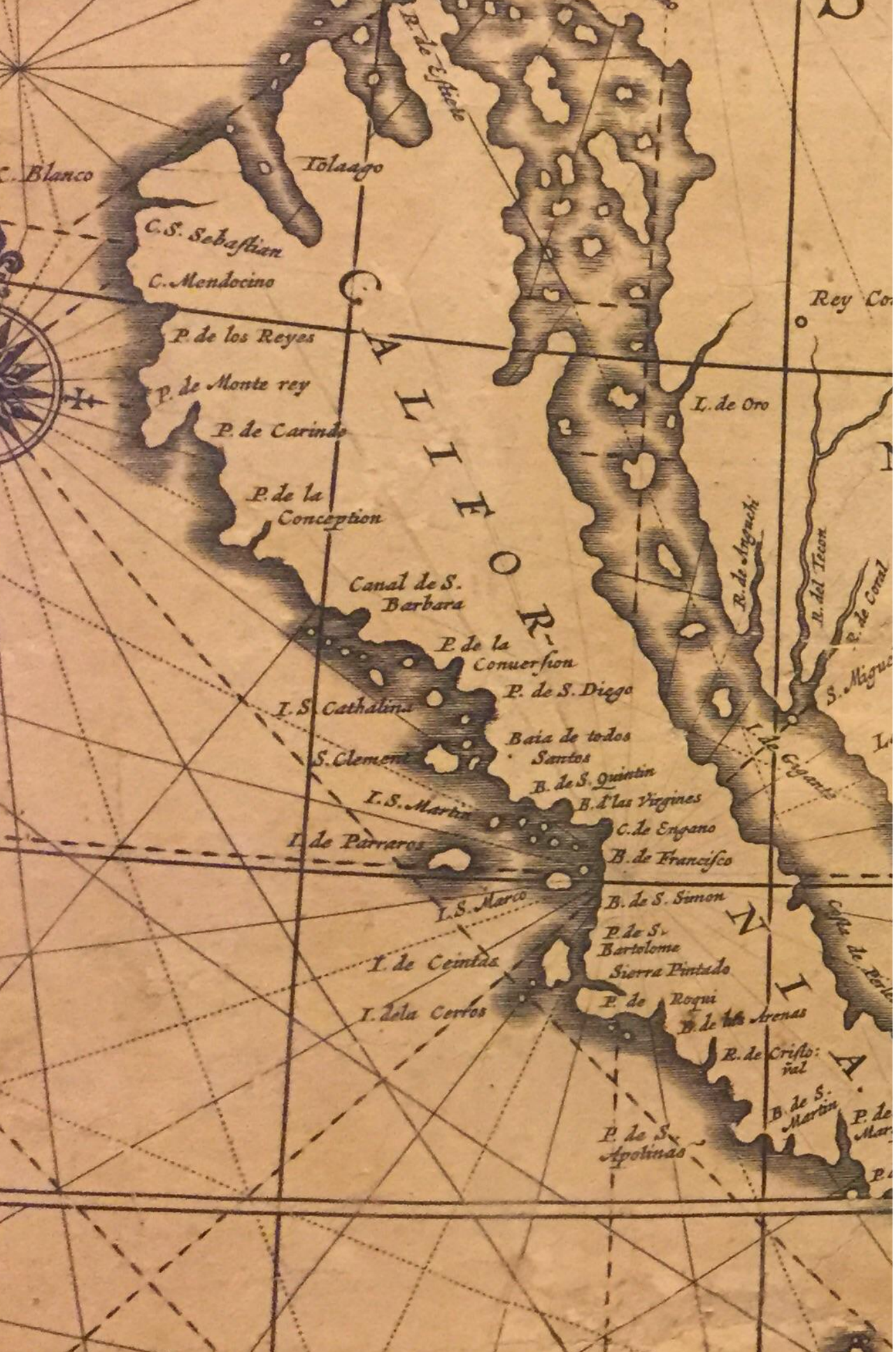
```
enum Collection {
    Phillumeny,
    Philately,
    Bibliophila,
}

fn thing_collected(collection: Collection) → String {
    match collection {
        Collection::Phillumeny ⇒ String::from("match"),
        Collection::Philately ⇒ String::from("stamp"),
        Collection::Bibliophila ⇒ String::from("book")
    }
}
```

A MATCH CONTAINS:

- The match keyword, followed by an expression
- The expression is *not* a predicate, unlike an *if* statement
- After this are a series of match *arms*, which consist of a *pattern* followed by code
- If the expression matches the pattern, the code is executed. Otherwise, the match continues

EXTRACTING A VALUE IN IF LET

```
enum Composition {
    Solo,
    Duet,
    Trio,
    Quartet,
    Combo(u32)
}
fn main() {
    let group = Composition::Combo(42);
    if let Composition::Combo(val) = group {
        println!("A group of {}", val);
    }
}
```

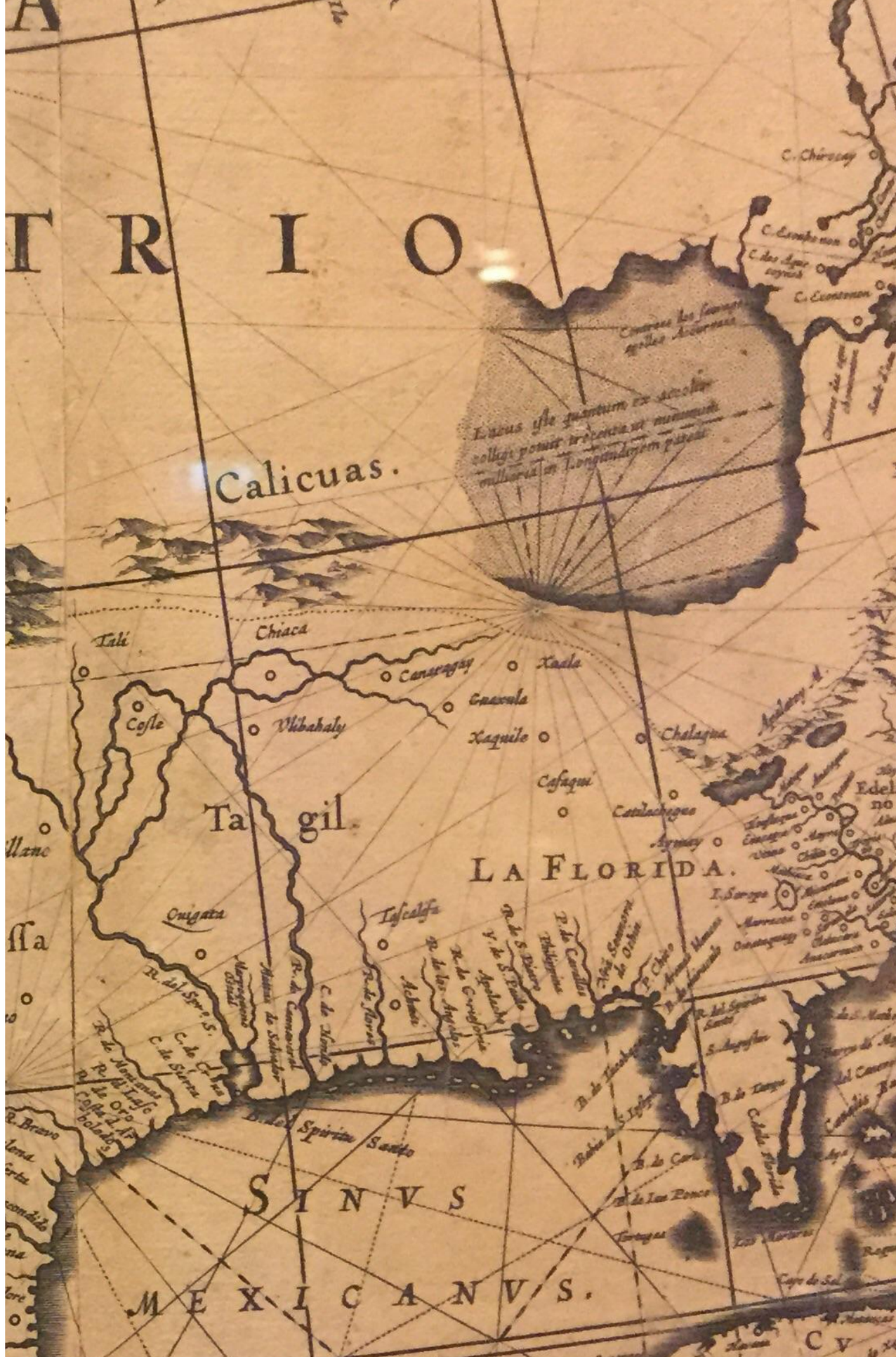


VECTORS

- A vector allows you to store more than one value in a single data structure that puts all the values next to each other in memory
- Vectors are strongly typed, and can only hold values of a single type

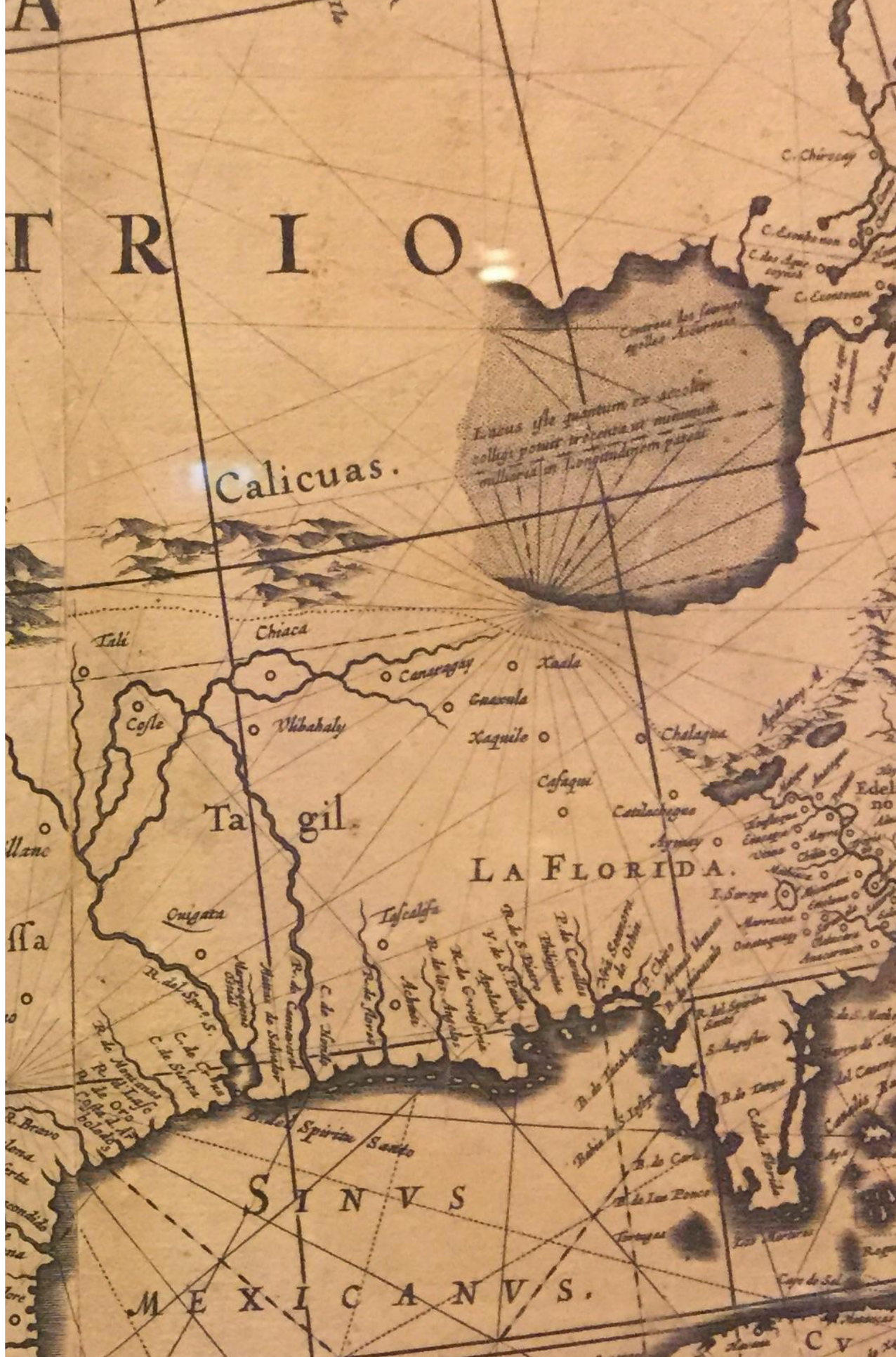
ACCESSING VECTOR ELEMENTS

- Elements of a vector can be accessed using bracket notation
- These elements are *references*
  - let a = &v[1];
- Indicies are not checked at compile time
- Accessing an out-of-bounds element will cause a *panic*



REMOVING ELEMENTS

- Add elements to the end of a Vector by using *push*
- Remove and return elements from the end of a vector by using *pop*



HOMEWORK

HOMEWORK

- This is very unfair! There's a test, and homework too?
- Please **BAIL OUT** on this homework if you have too much trouble with it!

- Take the interpreter we have so far (on the following slide), and add a "subtract" primitive to it.
  - You aren't going to be able to just use a different rust operator inside a copy of the "Add" arm, so spend some time thinking about this one
- If you legit can't do this in 20 or 30 minutes, drop me an email with your best attempt, and I'll review those and see how to make it all work out next week!
- The starter code is on the following slide…

```
#[derive(Copy,Clone)]
enum Primitive {
    Add,
    Multiply,
    Subtract,
    Number(i32)
}
fn evaluate(array: Vec<Primitive>) -> i32 {
    let element = &array[0];
    let mut iter = array.iter();
    iter.next();
    match element {
        Primitive::Add => {iter.fold(0, |total, next|
            total + evaluate(vec![*next]))},
        Primitive::Multiply => {iter.fold(1, |total, next|
            total * evaluate(vec![*next]))},
       Primitive::Number(val) => *val
    }
}
fn main() {
    let mut primitives = Vec::new();
    primitives.push(Primitive::Subtract);
    primitives.push(Primitive::Number(10));
    primitives.push(Primitive::Number(4));
    primitives.push(Primitive::Number(5));
    let result = evaluate(primitives);
    println!("result was {}", result);
}
```