Welcome to Statistics 750!

Christopher R. Genovese

Department of Statistics & Data Science

26 Aug 2025
Session #1

Plan

Introductions

Plan

Introductions

Motivation and Overview

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

Intro to the Fold Pattern

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

Intro to the Fold Pattern

A Look Ahead: The Even-Odd Problem

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

Intro to the Fold Pattern

A Look Ahead: The Even-Odd Problem

Appendix: Monoids

Announcements

- Please fill out **office hours poll**. (Treat dates as generic days of week.) https://www.when2meet.com/?31878095-c7DkK
- Special Office Hour times this week will be posted on Canvas. (Canvas is used mostly for announcements and some documents in special cases.)
- Email subject: [750]
- **Please bring your laptop to every class**
- **Reading this week**:
  - System Setup `https://36-750.github.io/course-info/system-setup/`
  - Thinking Languages Part 1 (gist for now, don't worry about details)
  - Interlude F Chapter 18 Section 1, material on monoids
- **Homework**:
  - Get a github account; send me your username in email.
  - **swag** assignment due Tue 9 Sep. Available on Canvas and github problem bank.
- Github invitations coming shortly, look for them.

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

Intro to the Fold Pattern

A Look Ahead: The Even-Odd Problem

Appendix: Monoids

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

Intro to the Fold Pattern

A Look Ahead: The Even-Odd Problem

Appendix: Monoids

Careful Thinking

Meaningful Abstraction

Good Design

Effective Practices

$\ldots \longrightarrow$

Careful Thinking

Meaningful Abstraction

Good Design

Effective Practices

Provable Correctness

Careful Thinking

Meaningful Abstraction

Good Design

Effective Practices

Efficient Performance

Careful Thinking

Meaningful Abstraction

Good Design

Effective Practices

Maintainability

Careful Thinking

Meaningful Abstraction

Good Design

Effective Practices

User Experience

Careful Thinking

Meaningful Abstraction

Good Design

Effective Practices

Extensibility

Careful Thinking

Meaningful Abstraction

Good Design

Effective Practices

Meeting Needs

Careful Thinking

Meaningful Abstraction

Good Design

Effective Practices

Fun

Careful Thinking

Meaningful Abstraction

Good Design

Effective Practices

Accomplished Goals

Motivation

- **Modern data analysis can be a complex business**
  Creating good software to manage this complexity has become an essential skill for statisticians.

- **Computing is taught *at the margins* in most statistics curricula**
  - Typical statistical computing courses focus on methods and algorithms for various concrete problems or on how to use current tools.
  - Students are expected to learn the practice of computing and software engineering organically during their research.
  - Typical feedback and incentives can obscure the benefits of building good software.

- **The organic development cycle for statistical software can limit correctness, clarity, and reusability.**

Motivation (cont'd)

- **Building efficient, elegant, reusable software increases our productivity and effectiveness**

  Good software engineering emphasizes:
  - Managing complexity
  - Communicating clearly
  - Building meaningful abstractions
  - Crafting well chosen solutions to problems
  - Obtaining good performance, reuse, and generalizability

- **Programming well is lots of fun**

- **Deep ideas can lead you to think about problems (math and stat not just computing) in a *new way*.**

  Exs: Algebraic Data Types, Categories, "Proof assistants"

- **What about AI???**

Core Axioms

- A broad, firm foundation in computing will pay off throughout your career
- The way to get better at programming is to **practice** programming
- Software design and programming practice are skills every statistician needs
- Revision is a critical part of the development process
- Having (at least a passing) understanding of multiple languages will make you a better programmer
- At the intersection of computing/mathematics/statistics are deep ideas and perspectives that apply far beyond computing

Key Skills and Goals for *Practice*

- Thinking about problems from new perspectives.
- Building and using meaningful abstractions
- Crafting elegant, maintainable, extensible software designs
- Choosing appropriate representations and algorithms
- Employing effective programming practices
- Developing efficient workflows
- Establishing Efficiency and Correctness
- Enabling generalization and reuse

Key Skills and Goals for *Practice*

By the end of this course, you should be able to:

- use types, algebraic structure, and several programming paradigms to reason about problems and code;
- develop correct, well-structured, and readable code;
- select appropriate algorithms and data structures for several common families of statistical and other problems;
- design useful tests at all stages of development;
- formally verify the correctness of programs and mathematical proofs
- effectively use development tools such as editors/IDEs, debuggers, profilers, testing frameworks, and a version control system;
- design and build a moderate scale software system that is well-designed and that facilitates code reuse and generalization;

Side goal: write small programs in a new language beyond R and Python. (Super-power languages: Clojure, Haskell, Rust, Racket; runners up: Ocaml, Scala, Julia).

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

Intro to the Fold Pattern

A Look Ahead: The Even-Odd Problem

Appendix: Monoids

Thinking Languages

We will emphasize more saliently the process of ***thinking about*** and ***designing*** our programs. To that end, we will develop in our discussions three **thinking languages** for talking about problems and designing solutions. For now, we will focus on the first two:

1. Thinking Language 1 is focused on using *types* to elegantly represent the concepts and entities that we are working with so and inform our design. It is spare and mathematical and allows us to express a variety of deep ideas. It is inspired by and descended from several beautiful (real) languages in current use, such as Haskell, Unison, Idris, Lean, and Agda.
2. Thinking Language 2 is an "imperative" pseudo-code for cleanly describing the steps in algorithms. It is relatively loose and informal putting clarity above syntax. It is inspired by scripting languages like Python, R, and Julia.

We will abbreviate these as TL1 and TL2. The Thinking Languages document introduces TL1 in detail.

Types

Types categorize data based on the how it is used and what it means.

Paying attention to types serves several valuable purposes:

1. Rich and explicit types make salient the concepts and *algebra* underlying your problem and your code, making it easier to think about the problem, design your solution, and reuse or generalize the code to other contexts.
2. Types help the compiler and IDE (and you) identify errors *before* your code is run, potentially improving productivity and correctness. The richer the type system, the more effective this is.
3. Explicit types let the compiler (and sometimes you) optimize your code to improve time and memory performance.
4. A sufficiently rich type system allows formal proof of code correctness (and other mathematical results).

Types make it easier to reason about and solve problems, to design complex software, and to produce performant code.

Two goals of a type system are to allow types that ***make the semantics salient*** and make ***invalid states unrepresentable***.

TL1 Types

We denote that a value `x` has a type `T`, by writing `x : T`. The `:` operator is read as "has type." A few important types in TL1:

- Built-in *primitive* types, including `Int`, `Nat`, `Double`, `Char`, `String`.

  We name types with an initial capital letter.
- `Void`, `()`, `Bool`
- The type `Type` of types, e.g., `Int : Type`, `String : Type`.

  We will make heavy use of **type variables**, e.g., `a : Type`. These can stand for any type or for types satisfying a constraint. We use *lowercase names* for type variables, often single letters.
- Tuple types. The types of tuples of arbitrary dimension with arbitrary type of components.

  Ex: `(a, b)` is the type of pairs whose first component has type `a` and the second component has type `b`. Similarly for `(a, b, c)`, `(a, b, c, d)`, and so forth.
- Function types. `a -> b` is the type of functions that take a value of type `a` as input and return a value of type `b`.

  We need to look at these a little more closely.

Function Types: A Closer Look

`a -> b` is the type of functions that take a value of type `a` as input and return a value of type `b`. `->` is a *right-associative* infix operator on types.

So: `a -> b -> c` means `a -> (b -> c)`, `a -> b -> c -> d` means `a -> (b -> (c -> d))`, and so on.

1. `a -> b -> c` is the type of functions that take two arguments, of types `a` and `b` respectively, and returns a value of type `c`.
2. `a -> b -> c` (aka `a -> (b -> c)`) is the type of functions that take an argument of type `a` and returns a *function* of type `b -> c`.

Functions in TL1 are automatically partial evaluated, so *these two interpretations are equivalent*. (We can always use parentheses in function types to disambiguate order in function types as with any other operator.)

What are these types?

- `(a -> b) -> c`
- `(b -> c) -> (a -> b) -> a -> c`
- `(a -> b) -> a -> b`
- `Nat -> (a -> a) -> a -> a`

Defining New Types

Types are more than just sets of values. We think of a type as constructed in particular ways, as characterized by specific semantics, and as goverened by particular *laws*.

In TL1, the primary method for defining types is the `data` declaration. This should be read as "data of type ______ has the form ______."

Consider a concrete example: *the type of a value that* ***may be missing***.

A potentially missing value may be missing or . . . not missing. In the former case, there is no value, and in the latter case we have an associated value.

Think of a potentially missing value as a box. We either have a box labeled "missing" or a box labeled "present" *with a value in it*.

Defining New Types

This is reflected in the `data` declaration:

```
data Maybe : Type -> Type where
  None : Maybe a
  Some : a -> Maybe a
```

`Maybe` defines a family of types, if `a : Type`, then `Maybe a` is a type. There are two ways to construct such a value:

1. To indicate "missingness" we use value `None`, and
2. To indicate presence of a value `x : a`, we use `Some x`.

The function `Maybe` is called a **type constructor**. The functions `None` (nullary) and `Some` (unary) are called **data constructors**.

Defining New Types

This is reflected in the `data` declaration:

```
data Maybe : Type -> Type where
  None : Maybe a
  Some : a -> Maybe a
```

`Maybe` defines a family of types, if `a : Type`, then `Maybe a` is a type. There are two ways to construct such a value:

1. To indicate "missingness" we use value `None`, and
2. To indicate presence of a value `x : a`, we use `Some x`.

The function `Maybe` is called a **type constructor**. The functions `None` (nullary) and `Some` (unary) are called **data constructors**.

There is a short form `data` declaration that is often more convenient:

```
data Maybe a = None | Some a
```

Here, `a : Type` is a type variable, and `|` is read as "or".

TL1 Function Definitions

Different ways to call a function

Python: `f(x, y, z)` Clojure: `(f x y z)` Haskell: `f x y z`

One might be more familiar, but the differences are not really big.

In TL1, we use the latter – operator – notation, like matrix multiplication.

Functions of multiple arguments are *partially evaluated* if given too few arguments.

```
max 4 2 : Int
max 4   : Int -> Int
max     : Int -> Int -> Int
```

TL1 Function Definitions

In TL1, all definitions are in two steps: a **type signature** and a **binding**.

Functions are defined by ***equations***, in mathematical style:

```
numItems : Int
numItems = 4

cube : Double -> Double
cube x = x * x * x

sumsq : Nat -> Nat                -- Computes $\sum_{i=1}^n i^2$
sumsq 0 = 0
sumsq 1 = 1
sumsq n = n * n + sumsq (n-1)

winsor : Int -> Int -> Int        -- Clip values to range
winsor threshold x
  | x >= threshold  = threshold
  | x <= -threshold = -threshold
  | otherwise       = x
```

TL1 Function Definitions

Defining equations can *match patterns* in the arguments. We can also define local variables (including functions) with `let` and `where` clauses.

```
maybe : b -> (a -> b) -> Maybe a -> b
maybe default f None = default
maybe default f (Some x) = f x

sum : List Integer -> Integer
sum [] = 0                          -- Empty list input
sum (x :: xs) = x + sum xs          -- Nonempty list starting with x

until : (a -> Bool) -> (a -> a) -> a -> a
until predicate action = go
  where go x | predicate x = x
             | otherwise   = go (action x)
```

TL1 Summary and Questions

We will use TL1 extensively to reason about problems, explore concepts, express computations, and establish results. It offers several advantages:

1. supports direct *equational reasoning*;
2. offers a rich set of concepts and constraints expressed through types;
3. enables formal verification of code (proofs of correctness);
4. encourages a fast and effective (type-driven) development cycle; and
5. facilitates clean and modular design.

In addition, working in TL1 will give us a new perspective on problems and programming.

It will take a little while to get used to, but it will be useful.

Questions?

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

Intro to the Fold Pattern

A Look Ahead: The Even-Odd Problem

Appendix: Monoids

Tales from the Loop

Focus on the loop in this function:

```python
def fib(n: int) -> int:
    if n <= 0:
        return 0

    a = 0
    b = 1
    for i in range(1, n):
        a, b = (b, a + b)

    return b
```

At the end `b` is the `n`th Fibonacci number.

We think of the loop as automating a sequence of instructions, but let's look at it in a different way.

Tales from the Loop

A different view of the loop:

```
loop with (a = 0, b = 1, i = 1):
    if i >= n:
        break
    restart with (b, a + b, i + 1)
```

Tales from the Loop

Let's repackage it a bit more formally:

```
def fib(n: int) -> int:
    if n <= 0:
        return 0

    def loop(a, b, i):
        if i >= n:
            return b
        return loop(b, a + b, i + 1)

    return loop(0, 1, 1)
```

We use a locally defined function. The `return loop` is just the `restart` from earlier.

The “Fold” Pattern

Here is a common computational pattern expressed in TL2:

```
1       accumulator = starting_value
2       for item in items
3           accumulator = update(accumulator, item)
```

This is called a **fold**. It yields the final value of the accumulator.

We have a collection of items, and we compute a value over the entire collection by successive updates for each item.

Line 1 initializes the accumulator. Lines 2–3 successively updates it. Here, `update` is often called the folding function, reducing function, or update rule. The final value of the `accumulator` is used at the end.

(Renaming `accumulator` to `state` reveals another view of this pattern.)

The “Fold” Pattern

Here is a common computational pattern expressed in TL2:

```
    accumulator = starting_value
    for item in items
        accumulator = update(accumulator, item)
```

This is called a **fold**. It yields the final value of the accumulator.

We have a collection of items, and we compute a value over the entire collection by successive updates for each item.

A fold is a general sequence *consumer*.

Here is a fold expressed as a *function* in TL1:

```
foldLeft : (a -> i -> a) -> a -> List i -> a
foldLeft _ start [] = start
foldLeft update start (x :: xs) =
  foldl update (update start x) xs
```

Ensure that you see the equivalence of these two formulations.

Some Useful Fold Variants

The "Scan" pattern:

```
    accumulator = starting_value
    collected = [accumulator]
    for item in items
        accumulator = update(accumulator, item)
        collected.append(accumulator)
```

This is called a **scan**. It is a fold that also collects all intermediate results, yielding the final values of `accumulator` and `collected`.

A scan is a general sequence *transformer*.

We can write the function `scanLeft` to implement this version of the Scan pattern. (`scanRight` does a scan for a fold from the right.)

```
scanLeft : (a -> i -> a) -> a -> List i -> List a
```

Example: Cumulative sums are computed with `scanLeft (+) 0 items`.

Some Useful Fold Variants

Left vs. Right Folds. The previous fold and scan operate from the left, taking the items in order and updating the starting value of the accumulator successively.

In contrast, a right fold starts from the right end, which has some advantages in certain circumstances.

```
foldRight : (i -> a -> a) -> a -> List i -> i
foldRight _ start [] = start
foldRight update start (x :: xs) =
  update x (foldr update start xs)
```

Note that the update functions given to `foldLeft` and `foldRight` have their argument orders swapped, with the accumulator parameter on the left or right, respectively.

Similarly, we can define an analogous `scanRight`.

Some Useful Fold Variants

Parallel Folds. Depending on the nature of the updates, folds need not be processed in either order.

In many cases, it is possible to do them in parallel, which allows both for improved efficiency and distributed computation over large collections.

This is a key part of the MapReduce idea.

The “Unfold” Pattern

Dual to a fold is an **unfold**, which generates a sequence from a `seed` and an unfolding function `ufn`.

Here is a simple TL2 implementation of the idea:

```
    collected = []
    while True:
        match ufn(seed):
            case None:
                break

            case Some((new_seed, value)):
                seed = new_seed
                collected.append(value)
```

As long as the unfolding function returns a non-trivial value, we update the seed and collect the value that was produced. This is a *left* unfold; if we used `collected.prepend` instead, we would get a right unfold.

The "Unfold" Pattern

Dual to a fold is an **unfold**, which generates a sequence from a `seed` and an unfolding function `ufn`.

Here's a simple TL1 implementation using Lists:

```
unfoldLeft : (s -> Maybe (s, v)) -> s -> List v
unfoldLeft ufn seed = loop (ufn seed) |> reverse
  where
    loop : Maybe (s, v) -> List v
    loop None = Sequence.Empty
    loop (Some (seed', value)) = value :: loop (ufn seed')
```

We could define an analogous `unfoldRight` as well by excluding the `reverse` step.

The “Unfold” Pattern

Dual to a fold is an **unfold**, which generates a sequence from a `seed` and an unfolding function `ufn`.

Here’s another implementation with a more flexible sequence type:

```
unfoldLeft : (s -> Maybe (s, v)) -> s -> Sequence v
unfoldLeft ufn seed = loop Sequence.empty (ufn seed)
  where
    loop : Sequence v -> Maybe (s, v) -> Sequence v
    loop collected None = collected
    loop collected (Some (seed', value)) =
        loop (append collected value) (ufn seed')
```

The “Fold” Pattern: Brief Exercises

Exercises:

1. Define a fold that computes the frequencies of items in the sequence, assuming the items are comparable. (R users can take the items in the sequence to be strings for simplicity.)
2. Define a fold that computes the five largest elements in a numeric sequence.
3. Define a fold that returns an element of a given index in the sequence.
4. Define a scan that computes an exponentially-weighted moving average of the values in a numeric sequence.
5. Define an unfold that computes the sequence of approximations by Newton’s method to the root of a smooth univariate function, stopping when the change (absolutely or relatively, your choice) is sufficiently small.

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

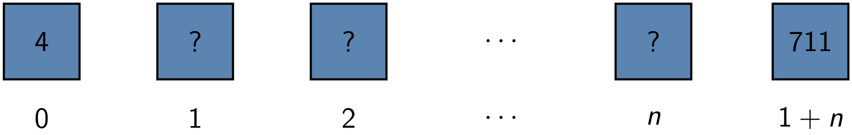
Intro to the Fold Pattern

A Look Ahead: The Even-Odd Problem

Appendix: Monoids

The Even-Odd Problem

In front of you are $n + 2$ boxes, each of which contains a natural number. All the boxes except the first and last are closed, so you can only see the first and last numbers. E.g.,



You want to find an *even-odd pair*, two adjacent boxes such that the first contains an even number and the second an odd number. You would like to do this by opening as few boxes as possible.

Plan

Introductions

Motivation and Overview

Thinking Languages and Types

Intro to the Fold Pattern

A Look Ahead: The Even-Odd Problem

Appendix: Monoids

Lists

Let $\mathcal{A}$ be a non-empty set. Then $\mathsf{List}(\mathcal{A})$, commonly denoted $\mathcal{A}^*$, is the set whose elements are *finite*-length tuples of elements from $\mathcal{A}$.

We can endow the set $\mathsf{List}(\mathcal{A})$ with structure by defining a binary operator $:: : \mathsf{List}(\mathcal{A}) \times \mathsf{List}(\mathcal{A}) \longrightarrow \mathsf{List}(\mathcal{A})$ that concatenates two lists. For example, $\langle 0, 0\rangle :: \langle 1, 0, 1, 1\rangle = \langle 0, 0, 1, 0, 1, 1\rangle$. This operator satisfies three **algebraic laws**, which you can confirm for yourself:

$$\begin{gathered}\langle\rangle :: \ell = \ell \\ \ell :: \langle\rangle = \ell \\ \ell_1 :: (\ell_2 :: \ell_3) = (\ell_1 :: \ell_2) :: \ell_3.\end{gathered}$$

where $\ell, \ell_1, \ell_2, \ell_3 \in \mathsf{List}(\mathcal{A})$. So, $\langle\rangle$ is an *identity* element for $::$, and $::$ is *associative*. Notice that $::$ is *not* commutative.

Monoids

A **monoid** $\langle \mathcal{M}, \Diamond, e \rangle$ consists of a set $\mathcal{M}$ equipped with a binary operator $\Diamond \colon \mathcal{M} \times \mathcal{M} \longrightarrow \mathcal{M}$ and a special element $e \in \mathcal{M}$ that satisfy:

1. $\Diamond$ is associative: $m_1 \Diamond (m_2 \Diamond m_3) = (m_1 \Diamond m_2) \Diamond m_3$ for every $m_1, m_2, m_3 \in \mathcal{M}$, and
2. e is an *identity element*: $e \Diamond m = e = m \Diamond e$ for every $m \in \mathcal{M}$.

$\Diamond$ need not be commutative, but if it is we call this a *commutative monoid*.

A special case we will use is an ***ordered monoid***, which is a monoid $(\mathcal{M}, \Diamond, e)$ with a partial order $\prec$ such that $x, y \in \mathcal{M}$ with $x \prec y$ implies

$$x \Diamond z \prec y \Diamond z \quad \text{and} \quad z \Diamond x \prec z \Diamond y,$$

for all $z \in \mathcal{M}$.

Monoids in Code

How might we represent a monoid in a program?

1. Wrapping data in an abstraction that makes it a specific monoid.
2. Subsidiary data for interpreting values in terms of a specific monoid.
3. Traits and generic monoid operators (dispatch handled by the compiler)

   . . .

We will see all these and more.

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$
3. Booleans with and or or. (What are the units?)

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$
3. Booleans with and or or. (What are the units?)
4. Natural numbers with max and ???

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$
3. Booleans with and or or. (What are the units?)
4. Natural numbers with max and ???
5. Subsets of a set with union or intersection

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$
3. Booleans with and or or. (What are the units?)
4. Natural numbers with max and ???
5. Subsets of a set with union or intersection
6. Multisets

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$
3. Booleans with and or or. (What are the units?)
4. Natural numbers with max and ???
5. Subsets of a set with union or intersection
6. Multisets
7. Functions $\mathcal{X} \longrightarrow \mathcal{X}$ with composition

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$
3. Booleans with and or or. (What are the units?)
4. Natural numbers with max and ???
5. Subsets of a set with union or intersection
6. Multisets
7. Functions $\mathcal{X} \longrightarrow \mathcal{X}$ with composition
8. Dictionaries

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$
3. Booleans with and or or. (What are the units?)
4. Natural numbers with max and ???
5. Subsets of a set with union or intersection
6. Multisets
7. Functions $\mathcal{X} \longrightarrow \mathcal{X}$ with composition
8. Dictionaries
9. Product monoids

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$
3. Booleans with and or or. (What are the units?)
4. Natural numbers with max and ???
5. Subsets of a set with union or intersection
6. Multisets
7. Functions $\mathcal{X} \longrightarrow \mathcal{X}$ with composition
8. Dictionaries
9. Product monoids
10. Dual monoids

Monoid Examples

What are some examples of Monoids? (Remember it's not just the set but the identity and operator as well.)

1. $(\mathbb{R}, +, 0)$
2. $(\mathbb{R}_+, \cdot, 1)$
3. Booleans with and or or. (What are the units?)
4. Natural numbers with max and ???
5. Subsets of a set with union or intersection
6. Multisets
7. Functions $\mathcal{X} \longrightarrow \mathcal{X}$ with composition
8. Dictionaries
9. Product monoids
10. Dual monoids
11. ...

THE END