Typst as a Language

Investigating the Typst programming language.
Justin Pombrio, 2024

Typst is a modern typesetting system, and a competitor to LaTeX. You can roughly divide it into two parts: it's a *programming language* glued to a *layout engine*.

The layout engine deals with:

- Margins
- Padding
- Subscripts
- Floating Figures
- Numbered references
- Justified text
- Right-to-left languages
- Hypenation points in words

The programming language deals with:

- Data representation
- Cyclic data
- Aliasing
- Mutability
- Garbage collection
- Control flow
- Functions
- Composability

This post investigates the *programming language* inside Typst. Since typesetting systems have very different requirements from most programming languages, Typst lacks some common programming language features but also includes unique features I haven't seen elsewhere. This makes it a fascinating case study.

(On the other hand, if you'd like to learn about Typst as a layout engine, here's an in depth post from that perspective.)

Overview

The most basic way to use Typst is as a markup language that's reminiscent of Markdown, plus the ability to write math inside `$`s:

```
= Heading

First paragraph, with _emphasis_
and *stronger emphasis*.

Second paragraph, with fancy math
$sqrt(5^2 - 4^2) = 3$.
```

⟶

Heading

First paragraph, with *emphasis* and **stronger emphasis**.

Second paragraph, with fancy math $\sqrt{5^2 - 4^2} = 3$.

All of this is *content*, which directly describes what to render to the screen. In this blog post, we'll focus instead on *code*. Here's an example with both content and code:

```
1 + 2 = #(1 + 2)
```

⟶ `1 + 2 = 3`

The + on the left is content—it appears in the output—while the + inside the code `#(...)` performs addition.

Ultimately everything you write in Typst is in one of three modes:

Markup Mode The default mode, that every file starts in. If you're in code mode, you can get back to markup mode with `[...]`.

Math Mode Enter math mode with `$...$`.

Code Mode Enter code mode with `#variable` or `#(expression)` or `#{multiple lines of code}`.

Since we're interested in the programming language of Typst, we'll mostly be using Code Mode.

Values

Let's talk about the different kinds of values Typst has.

Content

The most pervasive kind of value in Typst is *content*; it's what you get when you write in markup mode or math mode, and it's what is ultimately rendered.

Typst has a lot of builtin functions from content to content. For example, _underscores_ are a shorthand for a call to the function `emph`, which (by default) italicizes the content it's given:

```
_Underscores_ are a shorthand for
#emph[a call to emph].
```

⟶

Underscores are a shorthand for *a call to emph*.

So `emph` is a function from content to content. Notice that its argument is passed in square brackets `[...]` to mark the argument as markup.

I'll split the rest of Typst's values into *primitive values*, which cannot contain other values, and *compound values* which can.

Primitive Values

Typst has the standard primitives you'd expect: `none`, booleans, integers, floats, and strings. (You might be surprised by the strings since we already have content; I'll get to that in a minute.)

It also has some measurement types specialized for typesetting:

- **Length** (e.g. `2pt`, `3mm`). Used for things like the size of a font or the margins of a page.
- **Angle** (e.g. `10deg`). Used for shapes and rotations and such.
- **Fraction** (e.g. `2fr`), **ratio** (e.g. `50%`), and **relative** (e.g. `100% - 5mm`). Specifies how big something else should be compared to other things. I won't say more because it's more of a layout topic than a programming language topic.
- **Colors** can be specified in a variety of color spaces, including `rgb` and `hsl` and `oklab`. It's really great that perceptually uniform color spaces like `oklab` and `cielab` are built in.

Coming back to strings. Strings are different from content! Content has styling, such as font and size and color, while strings don't.

We haven't seen a string yet; how can you construct one? Since Typst wants to work as a convenient markup language, just putting double quotes in markup mode won't produce a string, it will produce quote marks. This covers the common case where you want to write down what someone said and have it be shown in matching quote marks:

```
"We're _almost_ there," she said.
```

⟶

"We're *almost* there," she said.

Instead, you need to be in code mode to construct a string:

```
#"This is a _string_."
```

⟶

This is a _string_.

Notice how the underscores in the string were rendered as-is. A string is a sequence of Unicode characters; only content has style.

There's something funny about this example. I told you that this is an example of a string (which is true), but that only content is rendered to the screen (also true). How then is this string getting shown?

Implicit Coercion to Content

What's happening is that the string is getting implicitly converted into content. This implicit conversion actually happened in an even earlier example too:

```
1 + 2 = #(1 + 2)
```

⟶ 1 + 2 = 3

The `1` on the left is *content*, while the `1` on the right is a *number*, and `1 + 2` on the right produces the *number* 3. Because the code block is inside content, the number it evalutes to is converted into content. Here's an example where this conversion is more visible:

```
3.0 = #3.0
```

⟶ 3.0 = 3

The conversion removes the redundant `.0`, while the content keeps it. (As a more extreme example, `3.x` is perfectly fine content, but `#3.x` is an error because `3` doesn't have a field called `x`.)

So strings and numbers are implicitly converted into content when they're used inside content. In fact, *all* values can be converted into content. Here's an array, and how it's printed as content:

```
#(3.0, 17in, "banana", rgb(50,
100, 150))
```

⟶

```
(3.0, 1224pt, "banana",
rgb("#326496"))
```

(Tangent: it's a little weird that 3.0 is displayed differently in an array than when it's standalone. Note the `.0` and the syntax highlighting when it's in the array. My guess is that Typst assumes that if you put `#3.0` in content you mean to render it as a number for readers, but if you put the array `#(3.0,)` in content you mean to render it for debugging purposes. And it's helpful when debugging to get syntax highlighting and to be explicit about decimal points in order to distinguish integers from floats.)

Compound Values

Let's look more at compound values like that array. Typst has just two kinds of compound values, which it calls *arrays* and *dictionaries*. These are completely standard data structures that almost every language uses. Unfortunately, what they're called hasn't exactly been standardized. Here's a table showing the names for three common data types in various languages:

Language	*Fixed Length Array*	*Growable Array*	*Hash Map*
Typst	-	array	dictionary
Python	-	list	dict
JavaScript	TypedArray	array	object
Java	array	ArrayList	HashMap
C++	array	vector	unordered_map
C#	array	List	Dictionary
Go	array	slice	map
Rust	array	Vec	HashMap
PHP	-	-	array
Ruby	-	Array	Hash
OCaml	Array	-	HashTbl
Racket	vector	-	hash

The precise definitions for those columns are:

Fixed-length array An array stored contiguously in memory, of a fixed size, with constant-time indexing.

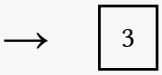
Growable array A array stored contiguously in memory, together with a *length* and a *capacity*. Also has constant-time indexing. The capacity is how big the array is, and the length is the number of elements that are currently used. If you append to this array and the additional elements fit within the unused capacity, they're filled in and the length is increased. However if the new length would be greater than the capacity, the array is re-allocated with a (multiplicatively) larger size. The fact that the backing array grows exponentially ensures that appending to it is amortized constant time.

Hash map A mapping from *keys* to *values*. The values are stored in a contiguous array in memory, where the index into the array is determined by the hash of the corresponding key. If the array gets too full, it's re-allocated with a multiplicatively larger size, ensuring amortized constant time insertion. Some languages, including Typst, require the keys to be strings. (This description doesn't say what happens if multiple keys have the same hash. There are a few ways to deal with that, all of which change the picture slightly.)

So Typst's `array` is a growable array and its `dictionary` is a hash map.

Typst arrays are written in parentheses and accessed with `.at()`:

```
#{
  let array = (9, 4, -1)
  array.at(1) + array.at(2)
}
```



You can also modify the array using `.at()`, like `array.at(0) = 17`.

(Tangent: this is a bit unusual. If you write `let x = array.at(0)`, `at` is a function call that returns the number 9. But if you write `array.at(0) = 17`, it's magically setting a value instead of calling a function. Most languages avoid using syntax that looks like function calls in the left hand side of assignments.)

You can also use destructuring in assignment like so:

```
#{
  let array = (9, 4, -1)

  // Shift elements left one, cycling around
  ((array.at(0), array.at(1), array.at(2)) =
   (array.at(1), array.at(2), array.at(0)))

  array.at(0) + array.at(1)
}
```

(The extra set of parentheses is needed because the assignment spans two lines. If it were all on one line, you could remove the outer parentheses.)

A dictionary is also written in parentheses, with colons to separate the key from the value:

```
#{
  let dict = (x: 0.5, y: 2.5)
  dict.x + dict.y
}
```

You can set a field of a dictionary using familiar syntax:

```
#{
  let dict = (x: 0.5, y: 2.5)
  dict.x = 5.5
  dict.x - dict.y
}
```

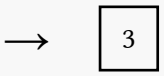
User-Defined Types

Most languages allow users to define their own types of values (e.g. `struct` or `class`). Typst doesn't support this, and I think that's a reasonable decision. User-defined types help structure complex programs, but there will be fewer of those in Typst than in a general purpose language.

Control Flow

Control flow is the rules for determining which code will run *after* which other code. Control flow in Typst is straightforward. It has runtime errors, which can't be caught. It has the usual control flow constructs `if`, `while`, `for`, `return`, `break`, and `continue`, all of which work in the standard way. There's a lot of questions in programming language design that are still up in the air, but these constructs we've figured out:

```
#{
  let array = (0, 1, 2, -100, 4)
  let sum = 0
  for x in array {
    if x < 0 { break }
    sum += x
  }
  sum
}
```



Short-Circuiting

There are another couple control flow constructs that you might not realize are control flow constructs: `and` and `or` (spelled as `&&` and `||` in many languages for historical reasons).

Say you want to check if an array is either empty or starts with `none`. You would likely write this:

```
array.len() == 0 or array.at(0) == none
```

If the array is empty, it's imperative that `array.at(0) == none` is not evaluated, because it would raise an error. So `x or y` first evaluates `x`, and if true it *never evaluates y*. Likewise, `x and y` evaluates `x`, and if false it never evalutes `y`.

This is a form of control flow. In fact, `and` and `or` together are as expressive as `if`! You can take any `if/else` statement:

```
if CONDITION {
  CONSEQUENCE
} else {
  ALTERNATIVE
}
```

and convert it into code that only uses `and` and `or` but behaves exactly the same:

```
let _ = CONDITION and {
  CONSEQUENCE
```

```
  true
} or {
  ALTERNATIVE
  true
}
```

(There are some fiddly details in there; understanding them isn't particularly enlightening. The important thing is to know that `and` and `or` can express everything that `if` can.)

Joining Content

I said that control flow constructs like `if` and `for` in Typst were completely standard. That's *mostly* true, though they have an interesting interaction with content.

You can use a `for` loop to perform mutation, like the earlier example that mutates `sum += x`. You can *also* use it to join multiple pieces of content together (taking an example from the docs):

```
#{
  let books = (
    Shakespeare: "Hamlet",
    Homer: "The Odyssey",
    Austen: "Persuasion",
  )
  for (author, title) in books {
    [#author wrote _#(title)_. ]
  }
}
```

→

Shakespeare wrote *Hamlet*. Homer wrote *The Odyssey*. Austen wrote *Persuasion*.

You can also make the whole body of the `for` loop be content by surrounding it by `[...]` instead of `{...}`:

```
#{
  let books = (
    Shakespeare: "Hamlet",
    Homer: "The Odyssey",
    Austen: "Persuasion",
  )
  for (author, title) in books [
    #author wrote _#(title)_.
  ]
}
```

→

Shakespeare wrote *Hamlet*. Homer wrote *The Odyssey*. Austen wrote *Persuasion*.

(In general, everything that expects `{...}` including `if` and `while` and functions can be given `[...]` instead.)

This joining behavior is similar to:

- Quasiquotation in lisps, where Typst code is to Typst content as lisp macros are to lisp code.
- `inline for` in Zig, where Typst code is to Typst content as Zig comptime code is to Zig runtime code.

That's all I have to say about control flow in Typst. It's overall, thankfully, very straightforward.

Abstractions

An *abstraction* is when you hide irrelevant details of something, so that people who use that thing don't need to think about them every time they use it. They only need to think about a smaller set of things, called its *interface*.

For example, a car is an abstraction. To use a car, you need to understand its interface, which includes things like using the gas pedal to accelerate and the steering wheel to turn. You do *not* need to know all the gnarly details about how the car works like VVT (variable valve timing), ABS (anti-lock braking systems), DCT (dual clutch transmission), APC (adaptive power control), or LSD (limited-slip differentials). Which is fortunate because one of those is made up.

(And yes, abstractions can leak. If your catalytic converter starts leaking, suddenly you need to know what a catalytic converter is, or find someone who does.)

Typst has two forms of abstraction, both of which are common in programming languages.

Functions

The first form of abstraction is functions. They're written with `let` syntax and generally work like you'd expect:

```
#{
  let add-one(n) = {
    n + 1
  }
  add-one(2)
}
```

As you can see in that example, functions don't need an explicit `return` (though you can use it for control flow). If the statements in a function produce multiple pieces of content and there's no explicit `return` statement, that content is joined together and implicitly returned:

```
#{
  let adoption-message(n) = {
    [You have chosen to adopt #n
kittens.]
    if n > 2 [
      That's a lot of kittens.
    ]
    [Remember to love and care for
your kittens!]
  }

  adoption-message(3)
}
```


You have chosen to adopt 3 kittens. That's a lot of kittens. Remember to love and care for your kittens!

There's no sensible way to use `return` statements in this function, so it's natural that Typst makes them optional.

There are a couple conveniences for functions. First, there are three kinds of arguments a function can take:

- Regular positional arguments. Defined like `let f(x, y) = ...` and called like `f(10, 20)`.
- Optional named arguments that take on a default value. Defined like `let f(x: 100) = ...` and called like `f(x: 101)`.

- An "argument sink" for functions that take any number of arguments. Defined like `f(..remaining-args)` and called like `f(1, 2, 3)`.

Second, there's a shorthand for passing content to a function. If you put brackets `[...]` after the arguments passed to the function, the content in the brackets is passed as the function's last positional argument. This simple trick is very convenient:

```
#{
  let repeat(times: 2, stuff) = {
    for i in range(times) {
      stuff
    }
  }
  repeat(times: 3)[
    There's no place like home.
  ]
}
```

There's no place like home. There's no place like home. There's no place like home.

One of the things mentioned in this section is going to turn out to be very important later, and essential for how Typst deals with typesetting.

Modules

Typst's other form of abstraction is modules. It conflates modules with files, which I think is reasonable for its aims. It's already dealing with all of typesetting, it should keep the language simple. There are 2.5 ways to "import" a module:

- `#import "foo.typ"` — puts `foo` in scope. Then you can write `foo.helper` to access what's defined with `let helper = ...` in "foo.typ".
- `#import "foo.typ": helper` — puts `helper` directly in scope, so that you don't need to prefix it with `foo.helper` every time you use it.
- `#include "foo.typ"` — inlines the *content* from "foo.typ". Importantly, style settings that were set by "foo.typ" don't contaminate the current file; you *only* get its content. (Much more on style settings later.)

This is *almost* well done, except for the fact that modules don't give you any way to mark items as public or private! Thus every `let` statement in the whole module is exported, even ones like `let horrible-fiddly-details-no-one-needs-to-know`. There's an open issue with a good discussion of the tradeoffs of different approaches, but no decision yet about how to move forward.

You can effectively make some items private if you're willing to wrap the whole module in a `let` that defines the public items:

```
// helper-mod.typ
#let (wow, double-wow) = {
  let priv-star = [☆]

  let pub-wow(msg)        = [#priv-star #msg]
  let pub-double-wow(msg) = [#priv-star #msg #priv-star]

  (pub-wow, pub-double-wow)
}

// main-file.typ
#import "helper-mod.typ": double-wow
#double-wow[The stars are out!]
```

Still, it's disappointing that every `let` in a module is public. This means modules aren't really an abstraction, since the implementation details leak out unless you go out of your way to use a pattern like the above to hide them.

Mutation

Let's talk about mutation.

Earlier, when I wrote the code `dict.x = 5.5`, it probably looked innocent to you. I, on the other hand, was laughing maniacally.

Mutation in programming languages is a bargain with the Devil. It's so easy to implement and seems innocuous, but then the Devil will:

- Create *cycles*, with `dict.x = dict`. This makes printing values harder (do you print `...` at a particular recursion depth, like JS does?), makes serializing and de-serializing values harder, and is the reason you can't use pure reference counting as a form of garbage collection.
- *Invalidate an iterator* by modifying an array while it's being iterated over: `for x in array { array.pop() }`. Is this undefined behavior that can result in memory corruption like in C++, or is there a runtime exception for it like Java's `ConcurrentModificationException`, or is it prevented by the compiler like in Rust, or does it have some fixed behavior?
- Modify a dictionary as it's being read in another thread. Is this undefined behavior like in Go, or guaranteed to produce an exception, or prevented by the compiler like in Rust?
- Use *global mutable variables*, which is now generally acknowledged as bad practice.
- Generally make it harder to understand and test code, because a function call could modify *just about anything*. Say you have a bunch of function calls in a row like `f() g() h() i()`. If `i()` isn't behaving the way you want it to, it could be because the behavior of `i()` depends on a value that's modified by a function `q()`, which was called by `p()`, which was called by `g()`. I like the term "*spooky action at a distance*" for this.

Let's try out some of these examples in Typst, to see how it deals with mutation.

Cycles

```
#{
  let dict = (x: 1, y: 2)
  dict.x = dict
  dict
}
```

⟶

```
(x: (x: 1, y: 2), y: 2)
```

Woah! That's unexpected. No cycle, instead just a copy of the original dictionary inside the new one.

Let's look at more examples and try to spot a pattern in what's going on.

Iterator Invalidation

```
#{
  let array = (1, 2, 3, 4, 5)
  for x in array [
    (#x, #array.pop())
  ]
  parbreak()
  [Final array len: #array.len()]
}
```

⟶

(1, 5) (2, 4) (3, 3) (4, 2) (5, 1)

Final array len: 0

So it did pop from `array` until it became empty, but at the same time the `for` loop iterated over the *original* array.

Global Mutable Variables

```
#{
  let x = 2
  x = 3
  let my-func() = {
    x = 5
  }
  //my-func()
  x
}
```

We're actually not allowed to call `my-func()`. It errors with the message "variables from outside the function are read-only and cannot be modified". So there are only sort of global mutable variables. You can mutate them outside of functions, and you can access them inside functions, but each function only sees the value the variable had when it was defined:

```
#{
  let x = 1
  let f() = x

  x += 1
  let g() = x

  f() + g()
}
```

Mutating Function Arguments

```
#{
  let modify-dict(dict) = {
    dict.x = 100
  }
  let dict = (x: 1, y: 2)
  modify-dict(dict)
  dict.x + dict.y
}
```

Ok, that's really strange. (Unless you're an R or Swift programmer, in which case this seems completely ordinary and you're wondering why anyone would think it was strange.) What's going on?

Value Semantics

Typst has what's called *value semantics*.

You can *think of* value semantics as always copying a value before passing it around:

- When you say `dict.x = dict`, the `dict` on the right is a *copy* of the original dictionary.
- When you say `for x in array`, the `for` loop gets a *copy* of the array, and iterates over that copy.
- When you pass `dict` to the poorly named `modify-dict` function, it receives a *copy* of `dict` and modifies that.

This gives a lot of advantages. It "foils the Devil":

- *You can't construct cycles.* All data is ultimately tree-shaped. This makes printing, serializing, and de-serializing data easier, and it makes pure reference counting a correct garbage collection strategy.
- *You don't need to deal with iterator invalidation.* It can't happen; you're always iterating over the original collection.
- *Multi-threading becomes easy.* The latest release of Typst came with multi-threading support, and it didn't sound like a Herculean effort. Contrast this to the multi year saga of removing Python's GIL.
- *Global variables aren't mutable from the perspective of a function.* For any given function, each global variable is a constant with a fixed value.
- *It's easier to reason about code.* You know that a function can't have side effects that influence how later functions behave. As a specific example, this means that the order in which you run test cases can't matter.

At this point you might be thinking that this sounds great, but it must be really expensive to copy every value each time it's used. Fortunately, while "copy every value each time it's used" is a correct mental model of Value Semantics, it's not the only possible implementation. Typst uses a more efficient "copy-on-write" implementation.

The downside of using copy-on-write is that the performance is unpredictable. As a programmer it isn't clear when you're going to accidentally initiate a deep copy. A value gets copied if and only if there is more than one reference to it, but it's not always obvious if that's the case.

An alternative is to explicitly distinguish uniquely owned values from references, like the Hylo language does. That is, every variable is *either* a uniquely owned value that you can modify freely (and no one else can see your changes because you're the unique owner), *or* a shared reference that you can't modify (unless you explicitly copy it to turn it into an owned value). This makes programs more complicated, but makes the places where copies happen explicit.

That's the tradeoff: copies are either implicit and difficult to predict, or explicit and easy to predict. The fact that Typst is a typesetting system has a couple effects on this tradeoff:

- Typst should aim for simplicity, because users should be thinking about typesetting instead of programming.
- Copies in Typst are likely to be cheaper than in most other languages. The largest data that's going to get passed around is content, but content is immutable so it never needs to be deep-copied.

Both of these differences make "implicit but difficult to predict" more attractive, so I think Typst was right to choose that option. (And I think Hylo, being a general purpose language, was right to choose the other option.)

Overall, I think value semantics is a *really* good fit for Typst.

Unique Features

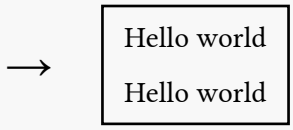
Most of what we've seen so far has been pretty run of the mill. But typesetting has some distinctive challenges that set it apart from other programming domains, and now we'll look at how Typst handles those challenges.

First Challenge: Tons of Style Options

You should be able to change the text font. As well as the font size, and the spacing between letters, and whether ligatures are enabled, and… well there are a lot of options. How can they all be organized and configured?

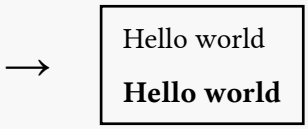
Typst solves this problem with one of the features we've already seen: arguments with default values. Take text styling. All text that's rendered in the document is made through calls to a function called `text()`, which is called implicitly:

```
Hello world

// Is shorthand for:
#text("Hello world")
```



There are a lot of options about how the text should be rendered, and they're configured by (optional) named arguments with default values. Thirty two of them. For example, the `weight` argument has a default value of "regular", but you can pass "semibold" to make it bolder (but not "bold" bold):

```
// Defaults to "regular" weight
Hello world

#text(weight: "semibold", "Hello world")
```



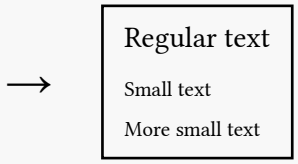
We talked about `text()`, but there are many other implicit functions with styles to be configured, like `heading()` for section headings and `enum()` for numbered lists. Using named arguments has an advantage over having a single enormous global set of styling options: it provides organization by grouping them by the function they apply to.

Second Challenge: Persistent Settings

What if we want to change the font size of the *whole* document? Or just *part of* the document? It would be beyond tedious to wrap every piece of text in the entire document in a call to `text()` with a `size` argument.

Typst solves this with a feature called `set`, which changes the default value of an argument:

```
Regular text

#set text(size: 8pt)
Small text

More small text
```



`set` can be used on any of the builtin "element" functions that render things, like `text()`, `header()`, `enum()`, and `image()`. It can't be used on user-defined functions, though.

`set` may look like it's simply setting a global parameter, but it's not! `set` never changes anything outside of its scope (the braces `{...}` or brackets `[...]` it's inside of):

```
#let disclaimer(org) = [
  #set text(size: 8pt)
  These claims have not been
evaluated by #org.
]

Zyverra instantly heals broken
bones.
#disclaimer[the FDA]
Buy Zyverra today!
```

Zyverra instantly heals broken bones.
These claims have not been evaluated by the FDA.
Buy Zyverra today!

See that the font size change only effected the disclaimer itself, not the text that came after it.

The fact that `set` doesn't "leak out" may seem minor, but it ensures a strong property. If you have some code like this:

```
Some text.
#some-function()
Some more text.
```

It is *impossible* for `some-function()` to change the font size (or any other property) of the text after it. This is an incredibly valuable guarantee that makes Typst easier to reason about as a user, easier to test, and easier to debug.

I've been hand-waving so far, but here's a precise way to think about `set`. It has three rules. The first rule is that you first partially evaluate the document, reducing everything down to `set` statements plus content. For example, this code:

```
#set text(fill: yellow)
#let scream = {
  set text(fill: red)
  [Aaaagh!]
}
#set text(fill: blue)
#scream Noooooo! #scream
```

partially evaluates to:

```
#set text(fill: yellow)
#set text(fill: blue)
#{
  set text(fill: red)
  [Aaaagh!]
}
Noooooo!
#{
  set text(fill: red)
  [Aaaagh!]
}
```

The second rule is that content is only affected by `set`s in enclosing scopes (surrounding braces `{...}` and brackets `[...]`). Thus `#set text(red)` won't apply to "Noooooo!".

The third rule is that if more than one `set` statement applies, the last one wins. (And if an argument is given directly, like `text(fill: green)[Yessss!]`, that takes priority over all `set` statements.)

Putting these together, you should be able to predict what that example is supposed to produce.

Here it is:

```
#set text(fill: yellow)
#let scream = {
  set text(fill: red)
  [Aaaagh!]
}
#set text(fill: blue)
#scream Noooooo! #scream
```

Third Challenge: Beyond the Builtin Parameters

What if you want to do something that isn't covered by one of the builtin parameters to an element function? For example, say you want top-level headings to be centered and in smallcaps.

Typst handles this with a feature called `show`. It lets you invoke a callback every time one of the bulitin element functions is called. Taking an example from a previous version of the Typst docs:

```
#show heading.where(level: 1): it
=> [
  #set align(center)
  #set text(
    13pt,
    weight: "regular"
  )
  #smallcaps(it)
]

= Smallcaps Heading
Putting your headings in smallcaps
makes them look sophisticated.
```

Breaking this down into pieces:

- `show heading` means we're going to bind a callback to be invoked on calls to the builtin `heading` function, which is called whenever you write `= Some Heading`, `== Some Heading`, etc.
- `.where(level: 1)` means that we shouldn't invoke the callback on *every* call to `heading`, only on the ones whose `level` argument is 1. (So it will be invoked on `= Some Heading` but not on `== Some Heading`.)
- `it => [ ... ]` is a closure (a.k.a. lambda expression, a.k.a. anonymous function). By convention, the argument is called `it`, but you can use any name you like. It's bound to the content returned by the original `heading()` function. Thus if you used the closure `it => it`, the `show` statement would leave the heading unchanged. Instead, our example's `show` statement renders the heading differently by setting it's alignment and weight, and putting it in smallcaps.

I described `show` statements as simply being callbacks, but actually they're a lot more complicated than that. They're more like macros, with a bunch of special cases for how they're invoked and applied. As an example of some of this complicated behavior, if you write `#show heading: it => [some text]` then `some text` comes out bold despite the fact that the callback discards its argument. I kind of want to criticize it for being overly complex, but despite having a PhD in this stuff I don't have an alternative that's unambiguously better.

`show` has the same scoping rules as `set`: it never affects anything outside of its scope. Again, this enables very powerful reasoning about your document. If you have some text that's too small, and you're wondering what changed the font size, the *only* possibilities are:

- There's an explicit `size` argument passed to the `text` function (duh).
- There's a call to `set text(size: ...)` in the current scope or an enclosing scope.
- There's a call to `show text` in the current scope or an enclosing scope.

Fourth Challenge: Tweak an Element

Say you want to put little flowers "✿" around each of your headings. There's no `heading(flowers: true)` parameter, nor are there `heading(prefix: "✿", suffix: "✿")` parameters, so you can't use `set` for this.

Let's try `show` instead:

```
#show heading: it => [
  ✿#(it)✿
]

== Flowers
make for soothing headings
```

→

✿

Flowers

✿ make for soothing headings

That sure didn't work! The trouble, I'm guessing, is that `it` is already in a `block()` which forces it to be on its own line, separating it from the flowers.

Let's try working with the text of the heading directly, which is accessible as `it.body` (because `body` is one of the arguments to `heading()`):

```
#show heading: it => [
  ✿#(it.body)✿
]

== Flowers
make for soothing headings
```

→

✿**Flowers**✿ make for soothing headings

Now we have the opposite problem: we *didn't* put the heading in a `block()`, so it isn't on its own line. Fixing that:

```
#show heading: it => [
  #block[✿#(it.body)✿]
]

== Flowers
make for soothing headings
```

→

✿**Flowers**✿

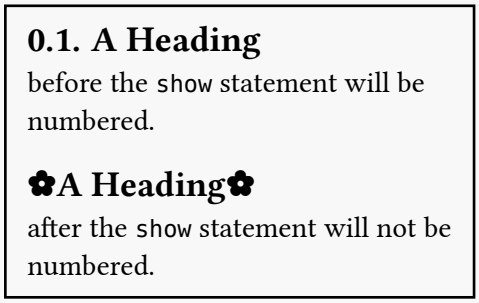
make for soothing headings

Looks good.

But now we've introduced a bug! It's hard to see because it hasn't been triggered yet. But say we enable numbering on our headings:

```
#set heading(numbering: "1.")

== A Heading
before the `show` statement will
be numbered.

#show heading: it => [
  #block[✿#(it.body)✿]
]

== A Heading
after the `show` statement will
not be numbered.
```

→



This is because we constructed the heading while ignoring `it.numbering`. The only *general* way around this sort of issue is to reconstruct all of the behavior of the `heading()` function in our `show` statement. This is the suggested solution on the Typst forums for how to make certain changes to headings or to term lists. This isn't ideal, as one of the most common reasons to reach for `show` is to slightly tweak how an element is rendered. It shouldn't be difficult to do so robustly.

Next I'll give suggestions for how Typst might be able to improve on this and some other problems.

Suggestions

Evolving a programming language is hard, because you want your users' existing programs to continue to work indefinitely while improving the language, and those goals are frequently in conflict. Fortunately I'm not a Typst developer, so I can ignore the realities of the situation and dream up whatever I like. Consider these suggestions in light of that!

More Powerful Show

We talked earlier about the fact that if you want to make a small tweak to how an element is rendered, like putting flowers around headings, you may need to write the formatting for that element from scratch, instead of being able to tweak the existing formatting.

As a second example that I actually encountered: if you want to make the terms in a term list not be bold, you have to say how to format each term/definition pair, including the spacing between them, even if you don't want to change that.

I had an idea for handling these situations by making it legal to write recursive `show` statements. But I spoke to a core Typst dev who suggested a much simpler approach. Allow `show` statements to apply not just to element functions, but to their arguments, like so:

```
show heading > body: it => [✿#it;✿].
```

This says that in every call to `heading()`, the `body` argument (call it `it`) should be replaced with `[✿#it;✿]`. Since it's only modifying the `body` and not the entire heading, it doesn't run into the issue with losing the `block()` wrapper that caused us trouble earlier.

Private Module Items

The fact that `#importing` a module lets you access every `#let` in the entire module is an abstraction leakage: some of those `#lets` are almost certainly implementation details. There's a discussion of this that covers the tradeoffs between the various possible solutions. I like the suggestion by one commenter:

Everything in a module remains public, unless there's *at least one* use of the `export` keyword. If so, only `exported` `lets` are public, and everything else is private. This has the advantage of being backwards compatible except for the introduction of the keyword, and it allows users to be lazy about public/private distinctions in places like modules local to their own projects where privacy doesn't matter as much.

(The commenter suggested the keyword `export`, but it could just as well be `pub`.)

`set` and `show` for User-Defined Functions

For this section we enter a vignette…

You're a geneticist in the year 2026, and you're writing a paper. Typst has caught on, and you're excited to use it for the first time. Your paper is going to mention a lot of genome sequences, and you're pleasantly surprised to find a ready made package with a `sequence()` function for displaying them. It has all sorts of formatting options:

- `type` ("DNA" or "RNA", default "DNA"). Determines whether thymine "T" or uracil "U" should be used.
- `order` ("sense" or "antisense" or `none`, default `none`). The order of the sequence: "sense" for 5′ to 3′; "antisense" for 3′ to 5′; `none` to omit the 3′ and 5′.
- `codon-sep` (string, default " "). The string used to separate codons. Use an empty string for no separation.
- `fasta` (boolean, default false). Use the standard `fasta` formatting. Overrides `order` and `codon-sep`.

You test it out and see that `sequence(type: "RNA", codon-sep: "-", order: "sense", "ATGTGTGGC)` produces:

```
5'-AUG-UGU-GGC-3'
```

Perfect.

(I know squat about genetics, so I'm hoping this is at least believable if not perfect.)

Then you get to work:

- Your paper has 137 genome sequences in it. You want all of them to be separated by dashes, so you write at the top of your paper `set sequence(codon-sep: "-")`. And they're all in "sense" order, so you add `set sequence(order: "sense")`.
- Your third section is about RNA, and it's in its own module, so you `set sequence(type: "RNA")` at the top of the module.
- You have an appendix that lists many sequences in `fasta` format for reference at the end, so you `set sequence(fasta: true)` in it.

All of this nicely separates content from presentation. The *same* sequence will be displayed differently depending on which section it's in. You just write `sequence("ATGTGTGGC")` and get (e.g.) `5'-AUG-UGU-GGC-3'` automatically.

This is good. You smile.

Exiting the vignette…

This story assumes that `set` works for user-defined functions like `sequence`, but today it doesn't. If this story were about Typst today, the geneticist would have had to either:

- Manually add the appropriate arguments to all 137 calls to `sequence()`, even if they were the same throughout the whole document, or

- Write a bunch of helper functions, one for every combination of arguments to `sequence`, and always use those helper functions instead of calling `sequence()` directly.

But once you've learned how `set` and `show` work, it's entirely natural to try to use them on a user-defined function like `sequence` instead of just on element functions. So my suggestion is to allow this. This is purely backwards compatible, as all current uses would be an (uncatchable) error.

Conclusion

I'm impressed by the design of Typst. Overall, its language is very minimal. Its use of value semantics makes it predictable and easy to debug. And it has a few features for dealing with the complications of typesetting like `set` and `show`. (And also `context`, which I didn't discuss in this post.)

And this is exactly what you want out of a typesetting system: a programming language that mostly gets out of your way, except to make a few hard things easier.