Technical University of Berlin
Faculty IV – Electrical Engineering and Computer Science

Typst

A Programmable Markup Language for Typesetting

Laurenz Mädje
Matr.-Nr.: [redacted]
Master's thesis, Computer Science
maedje@campus.tu-berlin.de

September 8th, 2022
First Examiner: Prof. Dr. Odej Kao
Second Examiner: Prof. Dr. Sabine Glesner
Advisor: Dr. Nicolai Stawinoga

Affidavit

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin,

....

Abstract

Markup languages are well-suited for typesetting of structured documents. By separating content from presentation, they are more automatable and flexible than their visual (WYSIWYG) counterparts. TeX-based systems, which trace back to the 1970s, are still the state of the art in this domain. Due to TeX's arcane macro system and poor error messages, these are hard to learn and use. More lightweight alternatives like Markdown and AsciiDoc fall short for complex typesetting while XML-based languages are too verbose for manual editing. This situation is unsatisfactory from a user experience perspective. For this reason, we present *Typst,* a new markup language with more expressive syntax and stronger computational foundations. By building on a type system with pure functions instead of macros, Typst eliminates many typical problems TeX suffers from. We show that Typst is highly automatable while being much easier to learn and use than current alternatives.

Zusammenfassung

Markupsprachen eignen sich gut für den Textsatz strukturierter Dokumente. Durch die Trennung von Inhalt und Darstellung sind sie automatisierbarer und flexibler als visuelle (WYSIWYG) Alternativen. TeX-basierte Systeme, die in die 1970er zurückreichen, sind in diesem Feld noch immer der Stand der Technik. TeXs obskures Makrosystem und schlechte Fehlermeldungen machen diese allerdings schwer zu erlernen und nutzen. Einfachere Alternativen wie Markdown und AsciiDoc sind für komplexen Textsatz ungeeignet, während XML-basierte Sprachen für die manuelle Bearbeitung zu umständlich sind. Diese Situation ist aus Sicht der Nutzer*innen unzufriedenstellend. Aus diesem Grund präsentieren wir *Typst,* eine neue Markupsprache mit klarerer Syntax und solideren programmatischen Grundlagen. Typst basiert auf einem Typsystem mit reinen Funktionen anstatt auf Makros und beseitigt dadurch viele typische Fehlerquellen von TeX. Wir zeigen, dass Typst zugleich sehr automatisierbar und deutlich einfacher zu lernen und nutzen ist als derzeitige Alternativen.

Table of contents

Chapter 1

Introduction

Today, there are two prevalent approaches to document formatting. The visual approach, also called WYSIWYG (*What You See Is What You Get*), and the markup-based approach. WYSIWYG tools let users work directly within the presentational form of the document. This has tangible benefits: They are easy to understand for most people and they provide immediate visual feedback. Markup, on the other hand, adds one level of indirection: Users write their documents in separate *markup languages* that mix text with commands. A compiler then transforms this markup into a presentational format. While this introduces upfront complexity into the workflow, it also provides rich grounds for automation. Consider, for instance, a text book that regularly contains blue boxes with extra details. In most WYSIWYG tools, these boxes would need to be created manually each time (or rather copy-pasted), making it very tedious to change the box's style later. With markup, the user can create a reusable abstraction for the box, decoupling its appearance from its occurrences in the document.

The most widespread kind of markup is *descriptive,* that is, the author inputs the document's *logical structure* rather than its concrete appearance. The latter is then defined by one or more *style sheets.* A prime example of this approach is XML [1], the *Extensible Markup Language.* XML is widely used in the publishing industry for storage, interchange, and formatting of books and articles. XML shines when there are large amounts of documents that follow the same *schema* (e.g., a series of books). By separating content from presentation, both can be freely changed at any point in time, with little manual effort. Alas, XML is quite verbose and thus not particularly suitable for being written by hand. Moreover, it quickly breaks down for more individual documents that do not following a strict schema. The popularity of lightweight markup languages like Markdown [2] shows the need for languages that are simpler to read and write.

TeX [3] is a markup-based typesetting system that was developed by famous computer scientist Donald Knuth for the purpose of typesetting his books on *The Art of Computer Programming.* Knuth was not satisfied with existing computer-based typesetting systems and decided it was time for a fresh start. [4] In the TeX language, formatting commands are realized as *macros.* A selection of macros for primitive formatting and low-level computation is built into the language, but users can also define their own macros. This extensibility sparked the development of many macro packages that improve or augment TeX in various ways; one of them LaTeX [5], a format that brought the idea of descriptive markup to the TeX world. TeX and its successors have enabled researchers and students to create papers and theses of high typographic quality while focusing on their research. It is not without problems though:

– Its macro-based programming model with many arcanely named primitives is difficult to grasp or build an intuition for. Writing anything but the simplest of macros is too complicated for most users, leaving them with the packages available on CTAN [6] (the TeX archive network).

– While TeX formats like LaTeX provide many things out of the box, even basic customization is often only possible by overriding specific macros. Finding the right macro to redefine and the right way to redefine it is often quite challenging. Moreover, this approach quickly leads to conflicts between different macro definitions.

– TeX's error messages are far from clear, and debugging more complex TeX code is extremely difficult. [7] While this could partly be improved through a better compiler, it is also partly a consequence of it being based on macros.

At a 1996 TeX user group meeting in the Netherlands, Knuth noted about the typesetting system *Troff* that "it was a fifth generation, each of which was a patch on another one. So it was time to scrap it." [4, p. 349] That is the exact situation we are in today, with layers of extensions and packages patched on top of the original TeX to make it do things it was not designed to do. Still, TeX is very widely used, not least due to its extensibility and its high output quality. TeX's fundamental promise is enticing. Not without reason have many projects been built on top of it. But as we will see in this thesis, TeX's dated design decisions fundamentally inhibit a great user experience.

We identify two central goals for future markup-based typesetting systems: First, they should be as user-friendly, understandable, and consistent as possible (*Simplicity*). Second, they should minimize the amount of manual labor necessary for document creation (*Automatability*). Specifically, they should maximize the amount of automation possible with a given amount of incurred complexity. To this end, we contribute a new markup language for typesetting called *Typst.* Typst sits in a very favorable spot on the automation-complexity spectrum: It offers a much better user experience than TeX while also being highly programmable. Compared to existing solutions, Typst has the following key novelties:

– *Seamless syntax for markup and code.* Typst mixes plain text, lightweight markup, and a complete programming language into a single consistent syntax. Markup and code seamlessly integrate and can be embedded into each other.

– *Strong computational foundations.* Typst includes a full programming environment built around pure functions. The language's type system makes it simple to handle layoutable content as a composable programmatic value. It also enables a compiler to produce user-friendly error messages with exact locations and call traces.

– *Composable styling.* Typst has a flexible styling system based on properties and transformations. With this system, users can style their whole document or parts of it. In particular, styles tightly integrate with content values, allowing for conflict-free composition.

– *Structural introspection.* Typst lets code inspect the document's structure and work with the final locations of elements on the pages in a controlled way. This forms the basis of the table of contents, section numbering, cross-references, and more.

We have implemented a compiler for Typst, demonstrating the feasibility of the language. The compiler can be used as a batch command line tool that converts `.typ` files to `.pdf` files. This is, however, not the primary envisioned usage scenario as the process of changing the source, recompiling, and reloading the PDF is still quite cumbersome. Instead, a web-based application is being developed in conjunction with the Typst compiler. This app provides side-by-side views of Typst source code with a rendering of the final document that instantly refreshes when the source is changed. While the web application is not a focus of this thesis, its requirements with regards to instant preview have considerably influenced the Typst language: All language features are designed in ways that allow the compiler to handle small edits incrementally instead of typesetting from scratch. The details about this strategy are part of M. Haug's master's thesis. [8]

This thesis is structured into nine chapters: Chapter 2 starts with a review of related work on markup-based typesetting, from the earliest languages to more recent attempts in the field. We then give a systematic overview over existing approaches to markup and styling. Among other things, we will discuss TeX-based systems and the solutions developed for the World Wide Web. Chapter 3 shifts the focus to Typst. It explains Typst from a user-facing perspective, presenting the syntax for markup and code and the available programming constructs. The following three chapters explore the Typst language in a layered model, starting at the *computational* foundations and then rising to the *presentational* and *structural* abstractions built on top of them. Chapter 7 explores the existing Typst compiler. In Chapter 8, we evaluate Typst's merits compared to other systems based on the aforementioned criteria of Simplicity and Automatability. Chapter 9 gives closing thoughts on this work.

This thesis is written in Typst and showcases the current compiler's capabilities.

Chapter 2

Background

There are two parts to a typesetting system. First of all, it should, of course, produce well-typeset documents. For instance, a high-quality typesetting system does not break lines one-by-one. Instead, it determines good line breaks for whole paragraphs at once (thereby producing far better justification results). Furthermore, it supports *kerning* (when letters move closer together, e.g. the 'T' and 'e' in 'Tea') and *ligatures* (when letters merge together, e.g. 'ffi'). To properly handle languages from across the globe, it performs contextual glyph positioning, substitutions and deals with right-to-left as well as bidirectional text. Ideally, it also finds good page break opportunities, preventing *widows* and *orphans* (lonely lines at the start or end of a page, respectively). As is made evident, the possibilities for improvement are endless and the complexities associated with them, too.

But the requirements do not end just yet. There is a second part: Aside from setting lines of type, the system should also aid its users in creating complex, structured documents with sections, headings, figures, tables, images, a table of contents, cross-references, and more. This means that the typesetting system has to understand the document's structure at least to a certain degree. This is where markup-based systems shine: They are much better at expressing a document's structure, as they can *show the structure,* whereas WYSIWYG systems show the final, typeset document. Markup-based systems also typically offer more powerful automations than WYSIWYG systems because markup facilitates abstraction within documents, either through *macros* or through *descriptive* markup. WYSIWYG systems, in contrast, are once again limited to what they can show the user.

In the next section, we review related work regarding markup-based typesetting. Despite some interesting open-source and commercial projects in development, this is not an active area of research. Relevant related work may therefore appear historic, but in many cases remains state-of-the-art—all the more reason to write this thesis.

In the two following sections, we will systemize the space of markup languages and styling systems. Specifically, we will discuss the two prevalent kinds of markup (*procedural* and *descriptive*) as well the two most important kinds of styling (*property-based* and *transformational*). This will form the basis of the remaining thesis and specifically of Chapter 8, in which we compare Typst to current alternatives.

2.1 State of the Art

The origins of markup-based text processing trace back to the 1960s. Among the first such systems were *DITTO*, *TJ-2*, *Runoff,* and *Roff.* But the concept only really took off when *Nroff* and *Troff* were developed at Bell Labs and incorporated into the Unix system. All the early systems, including Nroff, had in common, that they operated with fixed-width character sets. [9] In contrast, Troff (*Typesetter Roff,* pronounced *Tee Roff*) could typeset documents with dynamic fonts at different sizes, multi-column layout, and arbitrarily styled headers and footers. [10]

Troff and Nroff are source-compatible, meaning that users could create both fixed-width drafts and high-quality phototypeset prints from the same source document. [11] The source files consist of plain text intermixed with commands. Commands start with a period to distinguish them from text. The built-in commands provide basic actions like setting the font face (`.fp`), ejecting the current page (`.bp`), or centering the next N lines of text (`.ce N`). This form of markup is called *procedural,* as it directly instructs the computer how to proceed. Furthermore, users can define custom macros using the `.de` command. A macro gives a name to a common sequence of text and commands that would otherwise be duplicated in multiple places. [10] This led to the development of a multitude of macro packages that provided higher-level abstractions. [9]

Not many years later, in 1978, the famous computer scientist Donald E. Knuth started the TEX project. His goal was to create a system which he could use to adequately typeset his books on “The Art of Computer Programming.” [12] No existing system produced sufficiently high-quality output. With TEX, Knuth advanced the state of the art in computer-based typesetting, especially through its novel paragraph layout algorithm. In contrast to previous greedy algorithms that proceeded line-by-line, the Knuth-Plass algorithm [13] processes paragraphs as a unit and uses dynamic programming to find the best set of line breaks for the whole paragraph. TEX’s *box-and-glue* model, which underlies this algorithm, is quite flexible and enabled far higher-quality automated typesetting than previous approaches. However, TEX’s layout model also has its limits. In his 1993 publication “E-TEX: Guidelines for Future TEX Extensions,” Frank Mittelbach, a core member of the LATEX project, discusses a multitude of limitations with TEX’s layout implementation. These revolve especially around finding optimal page breaks and properly placing floating objects like figures. [7]

TEX has different syntax and different capabilities than Troff, but it retains its procedural nature: In TEX documents, users still mix their text with low-level control sequences instructing the system how to typeset the material. And like Troff, TEX supports macro definitions that combine text and primitives into a reusable definition. [3] Knuth envisioned that users would create their own personal macro collections with definitions useful to them. However, in practice, most people resorted to using off-the-shelf macro packages and formats—first and foremost LATEX, built by Leslie Lamport and initially released in 1984. [14]

LATEX took off because it strictly separated the concepts of content and presentation. This meant that most users could just focus on writing their document without worrying about typographical details or layout. Instead of marking the document up with commands, users would mark it up with *logical elements.* For instance, instead of adding commands to increase the font size and switch to boldface, LATEX users would write `\section{Introduction}` to get a nicely formatted section heading. The exact look would be defined by a LATEX document class or package. In his 1986 book “LATEX: A Document Preparation System,” Lamport initially motivates the idea with a mathematical document that contains many inner products. A user might want to use notation of the form (A, B) for inner products. In a WYSIWYG application, they would type this out whenever they needed such a product—and switching the notation later would be tedious. The idiomatic LATEX way is more flexible: The user defines a reusable command `\ip` and now writes `\ip{A}{B}` for an inner product. This way, the logical structure is separated from the visual representation and if they wished so, they could easily change this representation to another form like $\langle A|B\rangle$ later. [5] All of this was, of course, possible in TEX and Troff; after all LATEX builds on TEX. But only LATEX fully embraced the structural approach and provided a wide selection of built-in elements.

LATEX brought the idea of *descriptive* markup to the TEX world, but the idea itself is even older. In the early 1970s, Charles Goldfarb, Edward Mosher, and Raymond Lorie developed the *Generalized Markup Language* (*GML*) at IBM. [15] They were the first to propose embedding logical markers instead of processing instructions for a typesetter. Goldfarb describes procedural markup as inflexible: The same document cannot easily be prepared in different forms (e.g., draft and final print). He notes that editing with control words can be "time-consuming" and "error-prone" [16, p. 69] and states the following:

> *Markup should describe a document's structure and other attributes, rather than specify processing to be performed on it, as descriptive markup need be done only once and will suffice for all future processing.* — C. F. Goldfarb [16, p. 69]

GML is a *meta-language,* in which specific markup languages can be defined. While all markup languages defined in GML share the same general syntax, they differ in which elements are allowed in which places. These language definitions are called *models* or *document type definitions* (DTDs). [17] A model might define elements for paragraphs, headings, and quotes and then state that a paragraph can contain a quote, but that a heading cannot contain a paragraph.

Goldfarb published a paper about GML titled "A Generalized Approach to Document Markup" in 1981. [16] Just five years later, the ISO standardized GML as the *Standard Generalized Markup Language* (*SGML*). [15, 18] One of the changes from GML to SGML was the switch from a colon-based tag syntax to the `<tag>` syntax well-known from HTML. SGML was soon adopted by US government agencies, large companies, and the US military. [17] The key factor driving SGML's success was that it encoded documents as a single source of truth from which different products could be derived. [17]

In 1989, Tim Berners-Lee created the World Wide Web while working at CERN. For this pursuit, he needed a simple document format for *HyperText* that could be viewed on a variety of viewer devices. Since SGML was a suitable language with wide adoption, Berners-Lee created the *HyperText Markup Language* (*HTML*) as an SGML DTD. [19] HTML came with support for only a few semantic elements, most central among them the *anchor element* (identified by the `<A>` tag). This element makes up the *Hyper* portion of HTML by allowing one document to link to others—facilitating the creation of a web of documents. [20]

The second offspring of SGML is *XML*, specified by the World Wide Web Consortium (W3C) in 1998. It has a reduced feature set compared to SGML [21] (for example, it forbids unclosed tags and concurrent markup [22]). But it retains the most important aspect of SGML, one that HTML is lacking: The ability to define custom structural elements. This lets XML represent documents with much more semantic detail than HTML. [19]

With descriptive markup systems like SGML, HTML, XML or LATEX, a document's markup does not specify any particular appearance. This is where styling systems come into play. The most well-known of these styling languages is *CSS*, initially proposed by

Håkon W. Lie in 1994 [23] and standardized by the W3C in 1996. [24] While in the early days of the Web, the look of a document depended solely on the browser and the user's setting, increasingly document authors wanted to affect the look of their documents. A *Cascading Style Sheet* contains style rules that modify the appearance of HTML or XML documents. CSS is based on *selectors* and *properties:* The selectors define which elements' appearance is to be modified by the properties. Through the style *cascade,* a single document may receive styles from multiple style sheets. The other way around, multiple documents can use the same style sheet. [25] In combination, CSS style sheets provide a flexible and reusable way to style HTML documents.

However, CSS has the fundamental limitation that it can only set property values. It cannot create new elements (except through the very limited `:before` and `:after` pseudo-elements) and it cannot transform the structure of existing elements. As a result, many stylistic effects are impossible to achieve in CSS without modification of the HTML document. This leads to coupling of the document and the style sheet, somewhat countering the aforementioned benefits regarding flexibility and reusability. Transformation-based styling systems already existed at the time CSS was designed. However, the approach is unsuitable for the web as client-side styling with transformations would require the browser to download the whole document before starting to render it. [26]

One of the earlier transformation-based styling systems was *DSSSL* (*Document Style Semantics and Specification Language*). A DSSSL style sheet consists of style-specifications and transformation-specifications, both written with Scheme-like syntax. [27] Just like SGML, over the long term it was marginalized by a less general but simpler technology developed for the World Wide Web: The *Extensible Stylesheet Language* (*XSL*) which was published three years after CSS. XSL style sheets use the *XPath* [28] query language to select elements in the document on which template transformations should be applied. In the spirit of unification, XSL style sheets are themselves XML documents. [29] While simpler than DSSSL, XSL style sheets are still verbose and complex to write.

In the mid-2000s, a trend towards simplification emerged and *Markdown* saw the light of day. Markdown is a lightweight, descriptive markup language that is designed to be simple to read and write in plain text form. It has a fixed set of available structure elements, each with its own syntax composed of punctuation characters. This way, documents are much less verbose than ones written in SGML-based markup languages. A core idea of Markdown is that there are no invalid documents: Every text document is also valid Markdown, making the format especially beginner-friendly. [30] Markdown has built-in syntax for emphasized and strong text, links, images, tables, block quotes, code blocks, and a few other elements. For anything not covered by the built-in syntax, users can embed arbitrary HTML. [31] Markdown was widely adopted for simple use cases like README files, forum comments, or in-code documentation.

A similar mantra of simplification is followed by a multitude of languages that have emerged since. *AsciiDoc* [32], for example, supports basic macros, but otherwise aims to express most constructs directly in syntax. Similarly, *AsciiMath* [33] seeks to mimic real mathematical formulas as closely as possible in plain text. It sacrifices some of the capability of TeX for better readability and easier input. *Pandoc* [34] is a system that converts between many different markup languages and file formats, including

Markdown, LaTeX, Word, HTML, and more.

The TeX world has also seen modernization since the early days. XeTeX, first released in 2004, added full support for Unicode and modern font technologies to TeX. [35] LuaTeX, which arrived in 2007, embeds the *Lua* scripting language. [36] This makes it both easier to develop new abstractions and to extend existing systems. Many projects in the space develop a new front-end language while building on the established technological foundations of TeX (e.g., [37, 38, 39]). In principle, this is a sensible approach as TeX-based systems are powerful and produce great-looking documents. However, new systems following this approach also inherit some of the problems of TeX, including slow compilation, bad accessibility, and limited layout capabilities. For Typst, the complications and limitations of building on TeX weighed larger than the gain of not having to implement the core algorithms from scratch.

SILE (Simon's Improved Layout Engine) [40] and *Patoline* [41] are the opposite of the previous cases. They completely reimplement the typesetting process, improving on LaTeX in various ways: For example, both have good support for multilingual typesetting, SILE can layout on a baseline grid, and Patoline has a graph-based global layout optimization algorithm. However, both also more or less retain LaTeX's syntax. While we share the desire for beautiful typesetting with these two projects, Typst's primary focus is on designing a better user experience, including better syntax and a better styling system. SILE and Patoline do not seem to tackle these challenges.

Last but not least, work on LaTeX is still ongoing. LaTeX2$_\varepsilon$ was first released in 1994 and then, for the longest time, LaTeX stayed unchanged. [42] Although a third version of LaTeX was in continual development, the aims were too grand and the computers to slow—and more than twenty years later, in 2015, the LaTeX team shifted their strategy. Instead of developing an all-encompassing version 3, they are now in the process of "moderniz[ing] LaTeX through 'gentle refactoring.'" [42, at 20:30] This way, improvements and new features are available when they are ready instead of being batched up for a never-coming release. [42]

As part of this effort, in 2020, the L3 programming layer was merged into LaTeX2$_\varepsilon$. The LaTeX team had already envisioned this new programming layer in the 1990s, but at the time computers were not fast enough to handle it. Another LaTeX3 feature that was already merged in 2021 is the new *hook system.* [43] This system enables third-party packages to run setup code at the correct time in a well-defined manner, solving current problems with package conflicts. In 2022, at the time of this writing, the primary focus of the LaTeX project is to improve the accessibility of LaTeX's PDF output—specifically to create *Tagged PDFs.* LaTeX documents contain the structural information required for good accessibility (marked up headings, figures, captions, etc.) but this information is lost during typesetting. Improving accessibility is a multi-phase project that requires extensive changes to LaTeX's kernel. [42]

2.2 Markup

A document is more than an unconnected stream of words. Words form phrases and sentences, often separated by punctuation symbols like commas and periods. Sentences form paragraphs, which again build up into larger lines of thought—sections, chapters, and parts. Within those, we use headings, lists, and emphasis to structure our ideas. To present our thoughts, we use fitting typefaces, colors, and sizing; and we decide on single- or multi-column layout to create a coherent visual hierarchy. All these things happening on top of the stream of words are called *markup*. There are many different ways to mark up a document. These fall into different categories, the most important among them being *procedural* and *descriptive* markup. [44]

This thesis recurringly contains source code examples. The examples in this chapter present a variety of different markup and styling languages while later chapters mostly include examples in Typst. Many examples show both the source (to the left, on gray background) and the resulting output (to the right, on white background). When the source is not Typst, the left panel's top right corner designates which language it is in.

Procedural. The origin story of today's markup languages begins with *procedural* markup. Procedural markup is very natural for computers to consume. It consists of instructions to the typesetter, simple actions like *Set the font size, Skip one line,* or *Add two spaces.* [44] The individual actions are typically very primitive and often multiple actions are required to achieve basic effects. Repeating them over and over again would be very verbose. Therefore, many procedural markup systems support the definition of *macros* that encapsulate a sequence of text and actions. [44] A macro definition typically consists of the macro's name, sometimes optional parameters, and finally the replacement. In TeX, macros are defined with the `\def` primitive. [3] Below there are two examples of TeX macro definitions, one without parameters and one that repeats its parameter `#1` three times, with spaces in between:

TeX
```
\def\tu{Technical University}
Studying at the \tu ...

\def\thrice#1{#1 #1 #1}
\thrice{Hello!}
```

Studying at the Technical University…

Hello! Hello! Hello!

The capabilities of macros differ between different languages. TeX macros' parameter definitions are quite flexible; a macro can pattern match on the text after it. [3] For example, the macro below captures everything until the first comma in `#1` and everything between the comma and the first following exclamation mark in `#2`. It then formats the first part in boldface and the second part in italics. The final phrase "And the rest" is not captured by the macro and thus unaffected. (The percent signs start line comments preventing extra spaces between the fragments.)

TeX
```
\def\format#1,#2!{
  {\bf #1,}%
  {\it #2!}%
}

\format The first half, followed
by the second! And the rest.
```

The first half, *followed by the second!* And the rest.

When using macros to build more complex abstractions, the need for computation and encapsulation arises. To this end, the TeX language provides intermediate storage for different data types (*registers*), operations on these types, a grouping mechanism, and control flow primitives. Per type, there are 256 registers, numbered from 0 to 255.

For example, `\count0` refers to the first integer register while `\dimen10` refers to the eleventh dimension register and `\skip5` to the sixth skip (also called glue) register. There are multiple ways to interact with a register: To assign a value to it, one would write `\count10 = 5` and to add five to it, one would write `\advance\count10 by 5`. Any control sequence that expects a value of a type can also take the register: Both `\vskip 1cm` and `\vskip\skip10` are valid ways to add vertical spacing. [3]

How an abstraction is built is an implementation detail that should not be observable from the outside (aside from side channels like timing). Perhaps even more importantly, different abstractions should not interfere with each other. Most programming languages have facilities to support this kind of encapsulation (e.g., functions, classes, etc.) A bare macro system based on simple text replacement, however, does not (hence, the precedence problems with the C preprocessor [45]). TeX does have a mechanism for encapsulation called *grouping.* A group, written as `{...}`, isolates side effects occurring within it. Upon reaching the end of a group, TeX resets the values of all registers and configuration of things like font families to the state before the group. This allows users to freely work with registers within a macro definition without worrying about effects on the outside. Still, sometimes a side effect *should* transcend groups, for instance, when incrementing the page counter or a figure number. For this, TeX provides global assignments, written as `\global\advance\count0 by 1`. [3]

TeX also supports loops and conditionals, albeit in a somewhat roundabout way. Loops are implemented with repeated macro expansion. [3] The TeX code below computes and displays the smallest power of two larger than or equal to a number `\N` (here 1000). To achieve this, we first create a new counter named `\X`. Internally, this allocates one of the 256 count registers to `\X` (for instance, `\X` might expand to `\count72`). We then use the `\loop ... \repeat` construct to multiply `\X` by two while it is less than `\N`. We use the `\ifnum` command to perform this comparison. Finally, we display `\X` using the `\the` command.

TeX

```tex
\def\N{1000}
\newcount\X \X=1
\loop \multiply \X by 2
\ifnum \X<\N \repeat
The smallest power of 2
larger than \N\ is \the\X.
```

The smallest power of 2 larger than 1000 is 1024.

Descriptive. Today, *descriptive* (also called structural) markup is the prevalent form. Instead of specifying how an element *looks,* it expresses what an element *means.* [44] As an example, it might annotate a piece of text as being a postal address instead of noting that it is set in 11pt Times Italic. With descriptive markup, the concrete appearance of a structural element is determined in a separate step called *styling:* the transformation from descriptive markup to a presentational form. Notably, the same structured document may be transformed into different presentational forms with different style sheets. This makes descriptive markup very attractive for multi-platform publishing. [19]

The prevalent kind of descriptive markup is based on tags. This includes (S)GML, HTML, XML, and many more languages. A document in a simple GML model with headings, paragraphs, and quotes could have looked like the example below. Note that the markup only contains information-bearing text. Purely presentational text (e.g., the opening and closing quotes in the example) is generated during conversion to a presentational format. [16]

```
:h1.GML::h1.
:p.
This a paragraph with
:q.quoted::q. text.
::p.
```

GML

GML

This is a paragraph with “quoted” text.

How well descriptive markup can encode a document depends on the set of available structural elements. For instance, HTML cannot gracefully encode the semantics of many documents as it has a very limited set of elements. [19] While that is somewhat acceptable for websites, it is problematic for more complex pieces like educational books. For this reason, XML is very popular in the publishing industry. Among the most widely used standard sets of XML elements is JATS (for journal articles) [46] and BITS (for books) [47]. Apart from highly structured documents, XML is also widely used to encode and communicate data. The line between data and markup is fuzzy: For example, XML encoded travel data could be the source “markup” for a travel catalogue.

The second popular kind of descriptive markup builds upon macro-based procedural systems. In the case of LaTeX, a set of logical building blocks is encapsulated in the form of a *document class*. [5] While an article might be composed of a few sections with figures, plots, and mathematical material, a fiction book could contain a preface, multiple parts, chapters, and so on. The built-in classes are already somewhat customizable and some classes available on CTAN offer a bit more customization out of the box (e.g., the KOMA-Script ones [48]). Still, to gain full control over their document's appearance, users have to write their own document classes.

The example below shows a simple LaTeX document using the `article` document class and the `itemize` *environment*. Environments represent block-level constructs like lists or block quotes. They are surrounded by `\begin{..}` and `\end{..}` delimiters.

```
\documentclass{article}
\begin{document}

\section{Introduction}
This is a simple example.

\begin{itemize}
  \item The first item
  \item The second item
\end{itemize}

\end{document}
```

LaTeX

Introduction

This is a simple example.

– The first item

– The second item

The third category of descriptive markup consists of *lightweight* markup languages. These languages typically use punctuation symbols to structure the text with the minimal amount of intrusion. This makes reading and writing the markup easier. [30] On the flip side, most lightweight markup languages are quite limited in what they can do. However, as many are built on the foundation of a more powerful system like HTML or LaTeX, they often also support embedding syntax in the more expressive language. Below is a basic example showing some lightweight markup written in Markdown:

```md
# Markdown                                        MD
Text can be *emphasized* and
**strong**. And here is a [link].

[link]: https://daringfire...
```

Markdown

Text can be *emphasized* and **strong**.
And here is a link.

2.3 Styling

A styling system lets users customize the appearance of their documents. In Wysiwyg applications, styling a document directly affects the presentational elements. In very early visual editors centering a line was a one-time action that adjusted the spacing around the line just once. Removing or adding text later would leave the line uncentered. [49] In modern Wysiwyg systems, the text remembers that it is centered and stays that way. This is called *dynamic functionality* of a typesetting system: the "capability of the system to support change." [49, p. 32] For descriptive markup systems, styling affects the whole transformation of a document into a presentational format. A powerful styling system can transform the same source document into a wide range of useful outputs. Such a system has high dynamic functionality.

Selection. The first foundational mechanism of a styling system enables the user to *select* elements that should be styled with some *style rule.* In Wysiwyg applications, this mechanism is typically the mouse—users select a piece of text or part of the document and select a style from a menu. An important quality here is whether the application then retains the link between the text and the style rule or whether it simply applies the styles immediately. In the latter case, the text will not be updated when the users changes the style rule later, leading to a loss of dynamic functionality. [49] For descriptive markup languages, selection works differently. Here, the mechanism is usually a selection by element type. This way, the style sheet can set properties for all headings, lists or any other kind of element. More advanced styling systems also support context-dependent selection mechanisms, for instance to style emphasis differently within a heading than in body text. [49]

The most well-known and widespread selection mechanism is part of CSS. A CSS selector can select HTML or XML elements by tag, class, attribute value, and interaction state. Furthermore, CSS provides a variety of combinators for selecting elements within certain relationships (e.g., ancestors, parents, or siblings). [25] As CSS is designed for styling hypermedia documents that are viewed on a variety of devices, it also has built-in capabilities for adjusting styles depending on the viewer device. [25] For example, these *media queries* can customize the styles for devices that have a viewport with a width smaller than a given number of pixels (smartphones, tables, etc.) On the next page is an example of a CSS rule that colors emphasized text within strong elements in blue:

```html
<strong>
  Strong text that is
  also <em>emphasized.</em>
</strong>
```

```css
strong > em {
  color: blue;
}
```

Strong text that is also emphasized.

The second widespread selection mechanism for HTML and XML documents is *XPath.* [28] While the capabilities of CSS selectors and XPath overlap, the latter is more powerful: For example, it allows not only to select a child based on its ancestors but also a parent based on its descendants. CSS might gain a feature for selecting an element based on its direct children (but not arbitrary descendants) with *Selectors Level 4,* which is in draft state at the time of this writing. [50] TeX and its variants have no direct mechanism for complex selection.

Property-Based styling. At the foundation of most styling systems are configurable properties that "control the layout and appearance of document content." [49, p. 32] At a presentational level, they may configure the look of text (size, font selection, color, etc.) while at a structural level they might configure something like the numbering scheme for lists. Some properties might only be defined for certain elements (e.g. the numbering scheme) while others apply to a wide range of elements (e.g. the foreground color).

A property's *domain* defines which objects the property affects. [49] In CSS, all properties live in a shared namespace and can in principle be applied to any element. However, not every property actually affects every element. [25] The language syntax itself does not directly reflect the property domains, but if a property only affects one kind of element, this is sometimes reflected in the naming (e.g., `list-style-type` for styling lists). In TeX/LaTeX, there is no consistent system for configuring properties. Sometimes the user has to invoke a certain macro and sometimes they have to redefine it. Furthermore, the property domains are again only reflected through naming. The example below demonstrates the lack of coherence in the system. It shows *four* different ways to configure spacing between lines of text in LaTeX:

```latex
\onehalfspacing
\baselineskip15pt
\setlength{\baselineskip}{15pt}
\renewcommand{\baselinestretch}{1.5}
```

Configurable properties are naturally typed. For instance, a width property takes a length, a background property a color or fill and a rotation property an angle. Thus, a computational language with expressions and types is a good fit for expressing property values. CSS somewhat embraces this: Properties are composed from a common set of expressions and these expressions are organized in a simple type system consisting of lengths, angles, colors and so on. However, each CSS property can still have custom syntax. As a result, it is not possible to fully parse a CSS style sheet without knowledge of each individual property. [25]

Transformational Styling. In more flexible styling systems for descriptive markup languages, style sheets can also define custom transformations from source structures to arbitrary presentational elements. Such flexible mappings enable higher presentational fidelity without disrupting the source document's structure.

XSL is a transformational styling system for XML. An XSL style sheet consists of individual template rules. The two important parts of each template rule are the `match` attribute and the rule's body. The `match` attribute lets the XSL processor select elements by tag, attribute, or based on its relationship with other elements (through an XPath pattern). The XSL processor then applies the rule's body to the selected elements. [51] In the example below, the rule's body consists of two directives:

– The `<xsl:apply-templates />` directive instructs the XSL processor to also apply all defined template rules to the body of the matching element. [51] In this case, since there is no rule for `img` elements, the processor simply copies the image element to the output.

– The `<xsl:value-of .../>` directive extracts the `caption` attribute of the figure. The `select` attribute is once again an XPath expression. [51]

XML
```
<figure caption="A giraffe">
  <img src="giraffe.jpg" />
</figure>
```

XSL
```
<xsl:template match="figure">
  <div class="figure">
    <xsl:apply-templates />
    <xsl:value-of
      select="@caption" />!
  </div>
</xsl:template>
```

HTML
```
<div class="figure">
  <img src="giraffe.jpg" />
  A giraffe!
</div>
```

In some cases, the ideal definition of a structural element involves other structural elements. For instance, a glossary element might be transformed into a list in which the defined terms shall be emphasized. Ideally, the transformation can output an emphasis element which is then retransformed with the style sheet's rule for emphasis. XSL has no straightforward facility for this.

In LaTeX, transformational style sheets are implemented with the macro system and encapsulated in the form of document classes. Through the sheer power of TeX's macro system, LaTeX document classes have the capability to do almost anything. The price paid for this is that class files (and packages) are very complex to write. The example below defines a very simple class:

LaTeX CLASS
```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{myformat}[My Format]
\LoadClass{article}
\renewcommand{\normalsize}{\fontsize{10}{12}\selectfont}
\renewcommand{\section}{...}
\renewcommand{\maketitle}{...}
...
```

First, we declare that the LaTeX2$_\varepsilon$ format is required, what the name of the class should be and that the class extends the existing `article` class. [14] Then, we declare the font size and line spacing. For a real document class, we would also define font families, page margins, and so on here. Next, we renew the `\section` command. This allows us to customize how section headings print out. Redefining such a central command is complicated as it integrates with many different systems (section numbering, table of contents, cross-references). While it is possible to define a plain class that does not extend any of the existing classes, that class will lack many commonly expected commands like `\section` and `\maketitle`. Creating a class with all the expected bells and whistles from scratch is a lot of work.

Transformational styling systems like XSL can also perform tasks like extracting all section headings to build a table of contents. For tag-based descriptive markup languages like HTML and continuous output formats like web pages this is rather simple. For macro-based markup languages like LaTeX and paginated documents it is more complex. The first problem is that macro-based systems typically process the source file sequentially. While processing the beginning of a file, they have no idea what follows. Still, section information needs to be available at the beginning (the table of contents is typically at the start of a book, not the end.) The second problem is that the table of contents not only contains the section titles but also the page numbers of the sections. Thus, page layout must have already happened when the table of contents is built. LaTeX solves this problem by writing out the necessary information for the table of contents, lists of figures, citations and cross-references to a file in a first compilation and reading this file in a subsequent compilation. [5, 52] As a consequence, users often need to manually compile their document twice (and sometimes even thrice).

Chapter 3

The Typst Language

Typst is a new markup language for sophisticated typesetting. Like Markdown, it has built-in syntax for recurring elements like emphasized text, headings, or lists. But at the same time, it is also a quite capable programming language that supports conditionals, while and for-in loops, closures, and more. Notably, Typst is *not* a markup language with an embedded scripting language. It is *one* language that tightly integrates markup and code. Both can be embedded into each other and both can be handled as programmatic values. This allows Typst to provide powerful layout and styling systems, going way beyond what Markdown has to offer.

3.1 Markup

Typst's markup syntax follows Markdown in broad strokes—as that is anyway roughly what one would write when trying to format a plain text E-Mail:

– Text without any special symbols is just text.
– A blank line indicates a paragraph break.
– Text surrounded by a single star or underscore indicates strength (boldface, by default) or emphasis (italic, by default), respectively.

```
The first paragraph.

A second paragraph with
*strength* and _emphasis._
```

The first paragraph.

A second paragraph with **strength** and *emphasis.*

To structure a document into sections, subsections, and so on, Typst provides headings of different order. A first-order heading starts with a single equals sign, a second-order heading with two equals signs and so on. (Here we deviate from Markdown a bit because the hashtag is already used for something else in Typst.)

```
= The Typst Language
Typst is a ...

== Basic Markup
Typst's markup syntax ...
```

The Typst Language

Typst is a …

Basic Markup

Typst's markup syntax …

An unordered list consists of one or multiple lines prefixed with a hyphen. Lists can be nested, the structure is then determined through the indentation. List items can also contain paragraph breaks and arbitrary other markup. An ordered list is written by prefixing lines with a number followed by a dot. The number can also be omitted, leaving just the dot, to let Typst count through the items automatically.

````
*Shopping list:*
- Drinks
- Food
  - Bread
  - Tomatoes

*Plan:*
1. Write thesis
2. Finish studies
````

Shopping list:
- Drinks
- Food
  - Bread
  - Tomatoes

Plan:
1. Write thesis
2. Finish studies

There is built-in syntax for a few more common kinds of elements: The first is raw text that is used to include snippets of code. It is surrounded by an arbitrary but equal number of backticks and can contain all symbols that are otherwise special in Typst. When delimited by n backticks on each side, the code can contain up to $n - 1$ consecutive backticks. Raw text surrounded by at least three backticks can contain a language tag directly after the starting backticks. This instructs Typst to perform syntax highlighting.

~~~
The x86 register `rax` or
a ```java null``` pointer.

A minimal C program:
```C
int main() {
  return 0;
}
```
~~~

The x86 register `rax` or a `null` pointer.

A minimal C program:

```C
int main() {
  return 0;
}
```

Next are mathematical formulas, surrounded by dollar signs. Like raw text, such formulas can be inline with text or in “display” style as a separate block. In their display form, they are surrounded by an additional pair of square brackets inside the dollar signs.

````
Pythagoras: $a^2 + b^2 = c^2$.

By induction, we can prove that:
$[ #sum(k=1, n) k = (n(n+1))/2 ]$
````

Pythagoras: $a^2 + b^2 = c^2$.

By induction, we can prove that:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

A label, written as an identifier in angle brackets, identifies an element and can be cross-referenced by writing the at symbol (`@`) followed by the same identifier.

````
= Introduction <intro>
We begin with ...

= Conclusion
As discussed in @intro ...
````

1. Introduction

We begin with ...

2. Conclusion

As discussed in Section 1 ...

Additionally,

- A single backslash (`\`) indicates a forced line break and with an extra plus sign (`\+`) the line before the break stays justified,
- The single (`'`) and double quote (`"`) automatically turn into “typographical quotes” depending on the current text language,
- The tilde (`~`) indicates a non-breaking space,
- A hyphen with a question mark indicates a hyphenation opportunity (`hy-?phen`),
- The sequences `--` and `---` produce an en- (–) or em-dash (—), respectively,
- Three dots (`...`) produce an ellipsis (…),
- Any symbol that has a special interpretation in Typst can be escaped with a backslash to output it verbatim, e.g. `\=`,
- Line comments start with `//` and block comments are surrounded by `/*` and `*/`.

3.2 Blocks

Anywhere in markup, code may be embedded in curly braces, forming a *code block*:

```
The sum of 1 and 2 is {1 + 2}.
```

The sum of 1 and 2 is 3.

A code block may contain one or multiple expressions (separated by semicolons or line breaks). One such expression is a classic function call:

```
Lowest number is {min(7, -4, 3)}.
```

Lowest number is -4.

While mathematical functions like `min` are sometimes useful, most functions in Typst operate with *content values* instead. A content value can be created through a *content block*, written as markup surrounded by square brackets. For example, the `rotate` function takes an angle and content and returns content with an applied rotation.

```
{rotate(10deg, [_Angled_])}
```

Angled

A content block can not only contain arbitrary markup, it can also contain another code or content block. Blocks can be nested arbitrarily deep. Just like a function call or a content block, a code block is also an expression. The value resulting from the block is determined by *joining* the individual expressions in the block. See Section 4.3 for details on joining.

3.3 Functions

A function takes input values called arguments and computes from them a return value. In Typst, as in many other programming languages, a function call is written as the name of a function (or an expression that evaluates to a function), followed by arguments in round parentheses. Apart from standard positional arguments, Typst supports named arguments. These are written as a name, followed by a colon, and finally an expression (i.e. `fill: blue`). Positional arguments may be required, optional or variadic whereas named arguments are always optional. Function calls and content blocks are the foundation on which Typst is built. In fact, the markup introduced in the previous section is equivalent to function calls, as illustrated in Table 1.

Markup	Equivalent Function Call
Text	strong([Text])
Text	emph([Text])
```java null```	raw(lang: "java", "null")
$x + y$	math("x + y")
= Introduction	heading(level: 1, [Introduction])
- Bread - Tomatoes	list([Bread], [Tomatoes])
. Write . Finish	enum([Write], [Finish])
@label	ref("label")

Table 1: Markup and equivalent function calls.

Notably, the markup is only equivalent to standard library functions calls. A user cannot accidentally break Typst by defining a function named `list`. Typst's standard library ships with many useful functions for everything from text handling and layouting to computation, but users can also define their own functions (we will see how exactly in the next section). As it would become very cumbersome and symbol-heavy to wrap every function call in curly braces, Typst supports the following two pieces of syntactic sugar for function calls:

1. A code block containing a single function call can be shortened to use a hashtag in front instead of the braces. (This also works with identifiers and expressions that start with a keyword.)
2. Trailing content blocks in a function's argument list may be moved after the parentheses (with no spaces in between). Empty parentheses can be omitted in this case.

As such, the four following lines are all equivalent:

```
{list([Bread], [Tomatoes])}
#list([Bread], [Tomatoes])
#list([Bread])[Tomatoes]
#list[Bread][Tomatoes]
```

3.4 Bindings

In Typst, a computational variable is introduced with a let-binding. The variable is scoped to the containing code or content block, meaning it cannot be used after the end of the block. While a variable is live, it can be freely mutated.

```
{
  let count = 1
  count += 2
  count
}
```

```
3
```

When embedded in markup, identifiers and expressions starting with a keyword can be shortened with a hashtag just like function calls:

```
#let x = 1
#let y = 2
The sum of #x and #y is {x + y}.
```

The sum of 1 and 2 is 3.

Top-level let bindings live in the *module's* scope and can be imported from other files (see Section 3.7). A let-binding can be used to define a function simply by adding a parameter list after the identifier. The right hand side of the equals sign then defines an expression to evaluate every time the function is called. Notably, the right-hand side can also be a code or content block, as blocks are also expressions.

```
#let sum(x, y) = x + y
#sum(2, 3)
```

```
5
```

A function can capture variables from outside its definition. Such functions are called *closures.*

```
#let by = 3
#let multiply(x) = x * by
#multiply(2)
```

```
6
```

User-defined functions can also have named parameters. A named parameter is written just like a named argument, with the default value after the colon.

```
#let increase(x, by: 5) = x + by
#increase(3) \
#increase(3, by: 1)
```

```
8
4
```

For maximum flexibility, user-defined functions may also be variadic by defining an *argument sink.* In the example below, we define a `sum` function that sums up all its arguments. This example already uses *methods* and *for-in loops* which are explained in the following section. Note that we could also access the *named* arguments passed to the function by using the `args.named()` method.

```
#let sum(..args) = {
  let s = 0
  for x in args.positional() {
    s += x
  }
  s
}

#sum(1, 2, 3, 4)
```

```
10
```

3.5 Methods

To make coding in Typst ergonomic and convenient, Typst provides built-in utilities for manipulation of strings, array, colors, and more. However, implementing these as functions would quickly pollute the global namespace and lead to naming conflicts. For this reason, Typst implements them as *methods*. For example, the code below splits a string on whitespace (yielding an array), applies the `upper` function to each segment to uppercase it, removes the `"B"`, and then joins the resulting array with commas. This example also demonstrates how to create an anonymous function using arrow syntax.

```
{
  "a b c d"
    .split()
    .map(upper)
    .filter(c => c != "B")
    .join(", ")
}
```

A, C, D

In contrast to functions, methods can also alter the value they are called on:

```
{
  let array = (1, 2)
  array.push(3)
  array
}
```

```
(1, 2, 3)
```

Users of Typst cannot define their own methods. Methods are therefore only available on select built-in types.

3.6 Control Flow

Typst supports the classic imperative control flow constructs `if-else`, `while`, and `for-in`. A conditional is introduced by the `if` keyword, followed by a condition and then a block (code or content). Optionally, any number of `else if` branches can follow. Finally, the `else` keyword plus a block may follow. If the `else`-branch is omitted and no condition matches, the whole conditional evaluates to `none`.

```
#if 3 < 5 [
  Everything's fine.
] else [
  Math broke!
]
```

Everything's fine.

A `while` loop evaluates a block repeatedly while a condition is fulfilled. The resulting values from each iteration are *joined* like expressions in a block (see Section 4.3 for details).

```
#let i = 0
#while i < 3 {
  [Iteration #i. ]
  i += 1
}
```

Iteration 0. Iteration 1. Iteration 2.

A `for-in` loop iterates over a string, array (optionally with index), or dictionary (either with key and value or just value).

```
#let sum = 0
#for value in (2, 4, 6) {
  sum += value
}
#sum
```

```
12
```

The `break` and `continue` keywords can be used inside of loops to control the flow as in other languages. However, their semantics are adapted to improve their interaction with joining (also explained in Section 4.3).

3.7 Modules

As documents grow larger, users may want to split their project into multiple files. Typst has a built-in module system to support this. There are two ways to refer to another source file from within a source file:

- *Including:* The expression `include "path/to/file.typ"` evaluates the module located at the given path and yields a content value representing the whole file.
- *Importing:* The expression `import a, b from "path/to/file.typ"` evaluates the module and then makes the variables `a` and `b` available. For this to work, `a` and `b` must be defined with top-level `#let` bindings in `file.typ`. Writing `*` instead of `a, b` will load *all* top-level definitions of the module.

By default, paths are relative. They are only absolute to the root of the project if they start with a forward slash (`/`).

3.8 Set Rules

Many content-creating functions in Typst have different tweakable properties. For example, the `list` function has a `label` property that allows to configure how to label the items.

```
#list(label: [>], [One], [Two])
```

```
> One
> Two
```

A longer document may contain dozens of lists. Having to add the label argument to each list would quickly become tedious. It would also prevent the usage of the dedicated list syntax. Typst's solution to this kind of property-based styling are *set rules.* A set rule allows to set properties for all occurrences of a function (or its dedicated markup) for the remainder of the current scope. It is written with the `set` keyword, followed by a function, and then property arguments.

```
#set list(label: [>])
- One
- Two
```

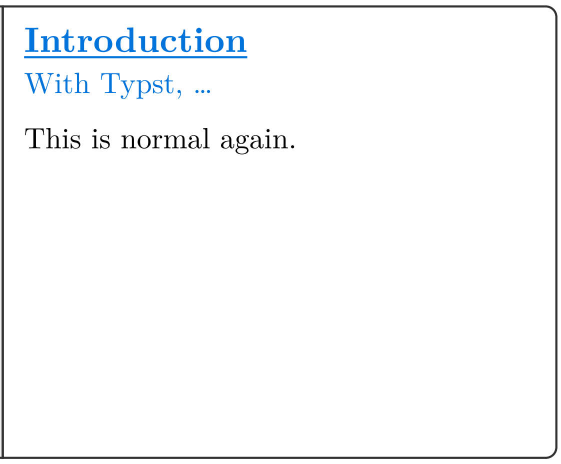
```
> One
> Two
```

Top-level set rules (written directly into the markup) can configure properties for the whole document. A document could, for instance, start with a list of set rules that configure the global style.

```
#set page(paper: "a4", margins: 3cm)
#set par(spacing: 1.4em, justify: true)
#set text(family: "Merriweather", size: 11pt)
...
```

However, set rules can also be local. This is best illustrated with an example. Below, we have a function that takes a content argument and styles it in blue and with underlined headings. Like a let binding, a set rule is only in effect until the end of the containing block (code or content). Therefore, the text after the function call is black again.

```
#let styled(body) = {
  set text(fill: blue)
  set heading(underline: true)
  body
}

#styled[
  = Introduction
  With Typst, ...
]

This is normal again.
```


Introduction
With Typst, ...

This is normal again.

Note that not every argument to a function is also a settable property. Properties include configuration that could reasonably be shared between multiple instance. This excludes for instance the content items of a list or the level of a heading. These are inherently tied to a single instance. For more details on set rules, see Section 5.3.

3.9 Show Rules

Settable properties provide many customization opportunities. However, there are limits to what they can do. Consider, for example, a section heading. A user may want to change the numbering scheme, switch the font, and add a decorative element to every heading. There cannot reasonably be properties for everything a user might want to change for each kind of element. That is where transformational styling with *show rules* comes into play. These let users define custom *recipes* that completely redefine the look of elements.

A show rule is written as:
- The `show` keyword,
- An optional binding, that is, an identifier followed by a colon (`:`),
- Then an expression that evaluates to a *pattern*,
- And finally the `as` keyword followed by an expression evaluating to content (the *body*).

```
#show element: heading as {
  set text(fill: blue)
  if element.level == 1 [📌 ]
  element.body
}

= Methods
This chapter discusses ...

== Environment
The importance ...
```

📌 Methods

This chapter discusses ...

Environment

The importance ...

In the example above, we define a show rule for headings. Therefore, the pattern is `heading`, which matches all individual headings. For each heading, the body is evaluated with the heading bound to the `element` variable. The original heading is then displayed as the result of said body. Notably, this is not a full replacement; the heading retains its semantic role and will still appear in a table of contents.

The heading object bound to `element` has multiple fields, including:
- A `level` field of type `integer` denoting the order (nesting depth) of the heading.
- A `body` field of type `content` encapsulating everything written after the equals signs.

The fields defined for each kind of element reflect the arguments that the element's constructing function takes. They provide the user with all the information needed to style the element to their liking.

Extending behaviour. Some elements have a complex built-in realization: The raw element, for instance, renders source code in monospace with syntax highlighting. A user show rule that customizes the raw element has access to the source text and the tag of the language to syntax highlight in (e.g., `rust`, `hs` or `typ`). Now, what if we wanted to display all code in rectangles with rounded corners, like in this thesis? Given just the source code and tag, we would have to reimplement syntax highlighting from scratch in Typst, which is obviously not an option. The crucial point is that, in contrast to the previous example, we want to *extend* the default behaviour instead of *replacing* it. This is exactly what *recursive show rules* enable. Instead of accessing the individual properties of the element, we use the whole element and place it into a rectangle. Typst then realizes the element recursively, this time with the default behaviour.

~~~
#show element: raw as rect(
  fill: rgb("#eee"),
  radius: 4pt,
  inset: 8pt,
  element,
)

```hs
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```
~~~

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Text replacements. Show rules can also transform plain text. For example, the show rule below italicizes all occurrences of the word “over”. Here, we use a *text pattern* instead of a function pattern:

```
#show word: "over" as emph(word)
As proven over and over again,
climate change is real.
```

As proven *over* and *over* again, climate change is real.

For even more flexibility, there are also *regular expression patterns.* This way, we can, for instance, easily place a box around each non-space character in a text.

```
#show c: regex("\S") as {
  rect(inset: 2pt, c)
}

This text is in boxes.
```

$\boxed{T}\boxed{h}\boxed{i}\boxed{s}$ $\boxed{t}\boxed{e}\boxed{x}\boxed{t}$ $\boxed{i}\boxed{s}$ $\boxed{i}\boxed{n}$ $\boxed{b}\boxed{o}\boxed{x}\boxed{e}\boxed{s}\boxed{.}$

For more details on show rules, see Section 6.2.

3.10 Wrap Rules

With `set` and `show` users can define flexible styling rules. Multiple rules can be combined into a style system by encapsulating them in a function. This approach scales up to whole documents. However, it would be inconvenient to wrap a document in a giant function call. This is were `wrap` comes into play: A wrap rule simply evaluates all content *after* it (until the end of the block or document), binds the result into a variable, and then evaluates its body with the bound variable. For example, below we have an `ieee` style system function defined in an external module. The `ieee` function takes some metadata about a paper and, as its last argument, the whole document body. With a wrap rule, we can capture the document body, bind it into an arbitrarily named variable (here called `doc`), and then pass it to the `ieee` function.

```
#import ieee from "papers/ieee"
#wrap doc in ieee(
  title: [Towards more ...],
  author: [Scientist \#739],
  abstract: [...],
  doc,
)

= Introduction
In recent years, ...
```

Towards more ...

Scientist #739

Abstract. ...

Introduction

In recent years, ...

Chapter 4

Computational Layer

Most markup languages with support for scripting allow the user to embed snippets of code in a separate programming language. These snippets interact with the document through constructs like JS's `document.write` or LuaTEX's `tex.print`. Typst pursues a different approach, in which markup and code are tightly integrated. Typst's built-in types are modelled around typesetting, with support for lengths, angles, colors, and most importantly: A content type capable of representing text, images, layouts, styling, and more. Typst adapts programming constructs to make working with content as ergonomic as possible. While configuring something in TEX often means redefining some internal macro, Typst favors functions that define composable abstractions with a clear interface. Conceptually, Typst is structured into three layers: *computational, presentational* and *structural.*

1. The computational layer forms the core of Typst. It defines a system of types and provides programming capabilities (most importantly, functions) which underlie the other layers. This layer shall be the focus for the remainder of this chapter.
2. The presentational layer provides the tools for displaying text and visual elements on pages, composing them into layouts, and styling them.
3. The structural layer defines structural elements and lets users define how these map to presentational elements. Human-created documents (a) are typically well-structured into sections, lists, etc. and (b) consist of recurring elements like headings, footnotes, figures, and so on. By embracing this fact, Typst can automate many tasks that users would otherwise need to perform manually.

4.1 Type System

A wealth of research has developed advanced type systems for programming languages. The same cannot be said for markup and styling languages. In TEX, macros consume and work with arbitrary token lists [3], while in HTML all attributes are plain strings. [53] In CSS, there are recurring property values like lengths and colors. But individual properties can still define ad-hoc syntax. [25]

Typst has a dynamic type system with few implicit conversions and a set of built-in types optimized for typesetting. This includes layout-specific types like lengths, angles, and colors, but, even more importantly, it includes a *content* type. A value of this type can hold everything from text and markup to images and layouts. It is represented as a tree of nodes that can also contain styling applied by set and show rules. Multiple content values can be *joined* (e.g., `[_Hello_] + [ World!]`). This combines the two underlying trees as subtrees of a new root. The content type is central in Typst: Full Typst files and content blocks result in content values and most built-in functions create, modify, or compose content. As described in Section 4.3, Typst's expression semantics are adapted to facilitate simpler composition of content values. Section 7.4 explains how content is represented in the compiler.

Typst is a dynamically typed language—any variable can hold a value of any type. Static type systems can be a very useful tool, primarily because they let programmers control a system's interface at its boundaries (i.e., at its API). This means that different systems can safely interact, and the programmer can rest assured that a compiler checked the points of interaction. In this line of thought, a static type system would be primarily useful to Typst package authors—as they would define such systems. Within a single document, however, the complexity of static typing outweighs it benefits. We should also keep in mind that most users of typesetting software like TeX are primarily neither typesetters nor programmers—they are students, researchers, authors, etc. It cannot be expected of them to learn and understand a complex static type system.

Conversions between different types are mostly explicit in Typst. For example, Typst does not implicitly convert a string into a number when trying to multiply it with a number (like JavaScript does, where `2 * "3" === 6`). However, it is also not as strict as Rust, which even forbids adding an integer to a float (`1.5 + 2` results in an error). Similarly to static type systems, these strict rules are useful in software development because they force programmers to handle error-prone operations early. In Typst's case, they add little value while increasing friction. Therefore, Typst performs implicit casts in a few cases (for example, integer/float addition or string to content conversion). However, it only performs such casts when the result is unambiguous. Note that the terms *weakly* and *strongly* typed are sometimes used to indicate the prevalence or absence of implicit conversions, respectively. However, the meaning of these terms is conflated and not universally agreed upon.

Table 2 lists Typst's 17 built-in types with examples and explanations. These types fall into the following categories:
- Typical basic data types: boolean, integer, etc.,
- Types specific to layout: length, angle, etc.,
- Collection types: string, array, dictionary,
- The content data type,
- A type for functions and one for captured variadic arguments.

Typst does not have a type hierarchy or polymorphism and users also cannot define their own types. There is, however, one thing all types have in common: They can be converted to content for user-facing display.

Type	**Examples and Details**
None	`none` Indicates the absence of any other value. Can be joined with every other value.
Auto	`auto` Many settable properties can take `auto` instead of a specific value to automatically select a good value. The exact meaning depends on the property.
Boolean	`false`, `true` Indicates whether something exists, is active, …

Integer	`12`, `-5` A 64-bit whole number, represented in two's complement.
Float	`3.14`, `1e-2` A 64-bit floating point number as defined in IEEE 754. [54]
Length	`12pt`, `1em`, `2em + 1cm` A physical length, distance etc. Apart from absolute units, there are font-relative units (`1em` is equivalent to the current font size).
Angle	`180deg`, `3.14rad` An amount of rotation.
Ratio	`50%` A ratio of some whole, mostly used to size an element in a layout.
Relative Length	`100% - 20pt` Combination of an absolute length and a ratio.
Fraction	`2fr` Describes how to distribute remaining space in a layout. For example, if element A is sized at `1fr` and B at `2fr`, then A will take one third of the remaining space and B two thirds.
Color	`rgb("239dad")`, `cmyk(80%, 9%, 0%, 32%)` A color in one of multiple color spaces.
String	`"Hello"` A UTF-8 encoded text string.
Content	`[*Hello*]`, `circle(radius: 1cm)` Composable representation of partially styled content that can represent text, layouts, and more. Can be constructed either through a content block containing markup or through functions like `image` and `circle`.
Array	`(1, 2, "3", false)` A heterogeneous, ordered collection of arbitrary values.
Dictionary	`(year: 2022, "with space": true)` A map data structure with string keys and arbitrary values.
Function	`(x, y) => x * y` A pure mapping from input values to a return value. It may originate from Typst's standard library, a let-binding, or an arrow function.
Arguments	`(..args) => ...` Arguments to a function, captured by an *argument sink.* On `args`, the individual arguments can be accessed dynamically. Alternatively, the arguments can be *spread* into another function call: `f(..args)`.

Table 2: List of Typst's built-in types.

4.2 Expressions

Most imperative programming languages differentiate between expressions and statements. Statements execute an effect while expressions produce an output value. In Typst, all code constructs are expressions. As we will see, this makes content composition more ergonomic. Typst's expressions fall into three groups.

1. Expressions that naturally yield a useful value: These are also expressions in most other languages. For instance, a binary `+` expression clearly evaluates to the sum of the left-hand and right-hand side.

2. Expressions that simply do not yield a useful value: While those are typically statements in other languages, in Typst they simply produce the `none` value. An example is the *let expression*, which binds a value to an identifier.

   ```
   #assert((let x = 2) == none)
   #assert(x == 2)
   ```

3. Expressions that yield a useful value if defined in the right way: This includes blocks, conditionals, and loops. While these are statements in typical imperative programming languages, they are expressions in more *expression-oriented* languages. For example, in Rust a block yields the value of the block's last expression and an `if-else` expression the value of the selected branch. [55] (Some languages have a separate *ternary operator* instead of defining if-else as an expression.) For and while loops are almost always either statements or produce a `none`-like value. In Typst, all three (blocks, conditionals and loops) are expressions and defined in a way that improves composability. The exact semantics are defined by a new concept called *joining*.

4.3 Joining

Two values may be *joined* to combine them into one. The semantics of this operation are similar to those of addition. It is defined for certain built-in types:

- Joining two string values concatenates them.
- Joining two arrays also concatenates them.
- Joining two dictionaries merges them.
- Joining two content values yields the combined content as if both had been written in the same content block directly one after the other.

In addition, every value can be joined with `none`, producing the value itself.

The *join* of a sequence of values $a_1, a_2, ..., a_n$ is defined as joining a_1 with a_2, then the result of that with a_3, and so on. If the sequence is empty, the join is `none`. The semantics of blocks and loops are now easily defined:

- A block yields the join of all evaluated expressions inside of it:

  ```
  {
    let x = "A, "
    x + "B, "
    "C"
  }
  ```

  A, B, C

This block joins `none` with `"A, B, "` and `"C"`. It is unproblematic that the let expression also contributes to the join as `none` can be joined with anything.

– A loop yields the join of each iteration's value:

```
#let i = 0
#while i < 3 {
  [Iteration #i. ]
  i += 1
}
```

Iteration 0. Iteration 1. Iteration 2.

In the first iteration, `[Iteration 1. ]` is joined with `none` (from the add-assignment), yielding just `[Iteration 1. ]`. Similarly, the second iteration yields `[Iteration 2. ]` and the third one `[Iteration 3. ]`. The content resulting from all three iterations is joined together and the result is what we see to the right.

The combination of an imperative programming style with expression-orientation distinguishes Typst both from popular imperative languages like JavaScript and from functional programming languages like Haskell. The imperative programming style with `if` and `for` is a natural fit for a markup language and relatively easy to pick up. It makes it simple to conditionally include a piece of content or generate content for something like a list of authors. At the same time, due to joining and expression-orientation, Typst's code block blurs the line between imperative and functional programming. It is unlike its imperative counterparts: Instead of being merely a container for multiple statements of code, through joining it essentially becomes *an operator with variadic arity.* Meanwhile, its syntax and scoping behaviour is typically imperative.

Control Flow. Typst also adapts the semantics of `break`, `continue`, and `return` to accommodate joining. When a control flow event occurs, the interpreter will stop further evaluation of code and content blocks, but already joined values will contribute to each nested block's output. Thus, in the example below, the `[#x]` part contributes to the resulting value of the block in the fifth iteration of the loop, whereas the `[, ]` part will be skipped.

```
#for x in range(100) {
  [#x]
  if x == 5 {
    break
  }
  [, ]
}
```

0, 1, 2, 3, 4, 5

An important detail is that the the control flow event only stops the interpreter from further evaluation of *blocks*. Each already started *expression* within any block up the evaluation stack will still be evaluated to its end. For instance, the `text` function is still fully evaluated even after the break within its argument.

```
#while true {
  text(blue)[
    My text is blue.
    #break
    And not green.
  ]
}
```

My text is blue.

Block Equivalence. Through joining, a content block can often be *flipped* into an equivalent code block and vice versa. Which way is simpler depends primarily on whether the block contains more markup or more code. A relevant difference is that the content block inserts a text space for every source line break whereas the code block does not. The examples below show the equivalence between the two kinds of blocks:

```
[
  #set text(family: "Signature")
  #city, the #date
  #v(20pt)
  #line(to: (4cm, 0pt))
]
```

```
{
  set text(family: "Signature")
  [#city, the #date]
  v(20pt)
  line(to: (4cm, 0pt))
}
```

4.4 Functions

Functions are at the heart of Typst. All elements without dedicated markup syntax are created through functions and even markup is backed by functions (see Section 3.3). Each Typst function is part of one of the three layers (computational, presentational, or structural). The computational layer defines the baseline semantics of functions. The higher two layers then add additional ways to interact with functions to automate document styling (see Section 3.8 and Section 3.9).

Arguments. Typst supports named arguments in addition to positional arguments. Named arguments are useful because typesetting is a very configurable task with many optional properties that have a good default value but still need to be changed in rare circumstances. Furthermore, Typst functions can deal with a variadic number of arguments. Capturing arguments with the `..args` syntax binds a value of type `arguments` into `args`. This variable has two methods, `.positional()` and `.named()`, that return an array with the positional arguments and a dictionary with the named arguments, respectively.

```
#let add(..args) = {
  let pos = args.positional()
  let named = args.named()
  pos(1) + named("to")
}

#add("ignored", 4, to: 2)
```

```
6
```

Value Semantics. Typst only has *value types*, not *reference types*. This means that, conceptually, each binding holds a unique, owned copy of its bound value. No two bindings can refer to and mutate the same underlying value. The below example illustrates this: Modifying the `second` array has no effect on the `first` array.

```
{
  let first = (1, 2, 3)
  let second = first
  second.push(4)
  [First is #first. \
   Second is #second.]
}
```

```
First is (1, 2, 3).
Second is (1, 2, 3, 4).
```

By extension, functions also always take their arguments by value and closures capture variables by value. Modifying a captured variable after the closure definition has no effect on the closure.

```
{
  let by = 3
  let multiply(x) = x * by
  by = 5
  multiply(2)
}
```

```
6
```

Naively implemented, value types would lead to many unnecessary deep copies when arrays and other composite structures are passed as arguments. The Typst compiler solves this through reference counting with a copy-on-write scheme. In the array example, `first` and `second` refer to the same storage until `second` is mutated. This approach is inspired by Swift. [56]

Functional Purity. Typst functions are *pure* as typical in functional programming languages. This means that they do not have side effects on any outer or global state and that they always return the same value given the same inputs.[1] This would not be possible without value semantics. Otherwise, a function's output could change when variables captured by reference change.

[1] In some cases, Typst functions interact with the environment. For example, the `image` function loads an image from the file system. Thus, its return value depends on the environment. In the framework of purity, this can be modelled as an implicit, immutable environment parameter that is passed through all function calls.

Purity is crucial for Typst because of the way it handles content. A piece of content can be initially created anywhere, stored in variables and data structures, and be used any number of times. Functions that modify the global state are at odds with this. Consider, for instance, a `figure` function that increments a global numbering counter. There is a difference between either *calling* that function twice or calling it once, storing the result in a variable, and *using* that variable twice. In the first case, the two figures have different numbers and in the second, they share the same number. Instead of relying on counters and impure functions, Typst employs *introspection* which is explained in Section 6.3.

Purity is also important from a performance perspective. A practical example is module caching: If Typst functions were impure and could modify variables in a module, the values in a module's scope could change over the course of one compilation. Still, at the start of the next compilation, the variables would again need to hold their initial values. This results in a compiler that has to evaluate *all* modules from scratch in every single compilation. In contrast, with purity, a Typst module is an immutable object that only depends on its source file and external files it loaded, imported, or included. In this setup, the compiler can retain modules whose source dependencies stay unchanged throughout multiple compilations. This solves a central problem LaTeX suffers from: As the execution depends on so many side effects (different counters, files, etc.), LaTeX has to reprocess all files in every compilation. Without caching, it becomes increasingly difficult to provide a responsive live preview for larger documents.

Note that in Typst there is no call statement that simply discards a function's return value. Due to joining, each function call in a code block contributes to the block's value. However, as Typst's functions are pure, a function is *only* called for its return value and ignoring it would render the whole call obsolete.

Chapter 5

Presentational Layer

While the computational layer forms the foundation on which all of Typst is built, the presentational layer provides the toolkit for bringing text and visual elements onto pages. It lies between the computational and structural layer. During layout, every element created in the structural layer gets mapped to a presentational representation (with the mapping being subject to the styling of the element, see Section 6.2 for details). To give users high design flexibility with little manual effort, the presentational layer provides:

- Building blocks like text, geometrical shapes, and images.
- Mechanisms to compose these building blocks into hierarchical layouts.
- A principled styling system for configuration of all kinds of visual properties—for both the building blocks and the layouts.

5.1 Foundations

A layouted document consists of quite basic elements. What is ultimately visible on the pages is text (in the form of positioned *glyphs*), basic shapes like lines and curves, and images.

Text. The most fundamental building block of any document is text. Reflecting this, whatever a user starts writing into a Typst document (with the exception of a few reserved characters) is interpreted as text. Despite being so essential, correct text handling is an incredibly complex task. To handle the rich variety of human writing systems and languages, Typst's text stack supports:

- Unicode: Through Unicode, Typst can handle a variety of scripts within a single file. This includes, among others, עִבְרִית (Hebrew), عربي (Arabic), देवनागरी (Devanagari), and 🌳 (Emoji).

- Text shaping for international scripts: In languages like Arabic, characters can join together and take different forms depending on the context.

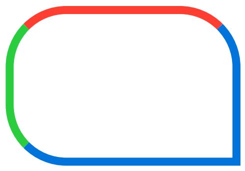
- Bidirectional text: When mixing text in scripts written from left to right and right to left, the text is reordered according to the Unicode Bidirectional Algorithm [57] to be fluently readable.

- Unicode-aware line breaking with hyphenation and justification: A line break cannot be inserted before every character (e.g., not before a colon). Besides, good justification sometimes requires inter-word breaks at syllable boundaries. Rules for syllable segmentation differ by language.

- Font selection with fallback: No font supports all of Unicode. To still render as much text as possible, Typst finds alternative fonts for unsupported characters.

Shapes. The second fundamental building block are geometric shapes (rectangles, ellipses, and bézier curves). These form the basis of underlines, rules, table borders, and more. Typst has built-in functions for different shapes. The following example shows how to create a rectangle with corner radii and differing stroke colors:

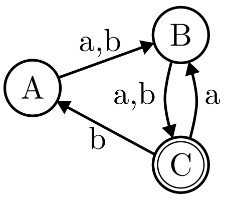
```
#rect(
  width: 3cm,
  height: 2cm,
  radius: (left: 75%, top: 75%),
  stroke: (
    left: green,
    top: red,
    rest: blue,
  ),
)
```



Through support for Beziér curves, even text could be reduced to curves. However, this is undesirable as it prevents viewers of the document from selecting and copying the text.

Images. The third building block are images. Typst supports raster images (PNG, JPG, GIF) and vector images (SVG), which don't lose precision when zoomed in. These are especially useful for scientific illustrations.

```
#grid(
  columns: (55%, 1fr),
  gutter: 5pt,
  image("beach.jpg"),
  image("dfa.svg"),
)
```



5.2 Layout

Building blocks are only useful if there is a way to arrange them on the page. Typst (like web browsers and UI toolkits) provides essential compositions. By combining these, users can produce a wide variety of layouts. Below, we discuss a few of the built-in compositions. Figure 1 illustrates alignment, stack, and grid layout.

Paragraphs and Flows. Paragraphs and flows are the primary way Typst combines multiple elements. A paragraph arranges text, images, and other elements horizontally (from left to right or right to left depending on the selected language). A flow arranges paragraphs and other block-level elements vertically (from top to bottom). Typst automatically creates paragraphs and flows as necessary when multiple elements are given where one is expected.

Alignment. Often, an element takes less space than what is available on the page. With the `align` function, the element can be placed to one of nine positions in the larger space —into one of the four corners, at one of the four sides or into the middle.

```
#align(right + bottom)[Aligned.]
```

Aligned.

Stack and Grid. Apart from paragraphs and flows, stack and grid layouts are the primary two ways to combine multiple elements into one. A stack layout places multiple items along one axis. Optional spacing between the elements can be either absolute, relative (to the full space), a mix of the two, or fractional. Fractional spacing distributes

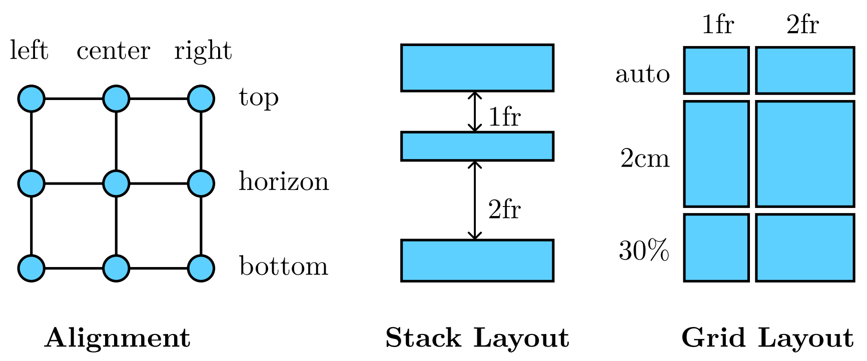


Figure 1: Illustrations of different layout compositions.

the remaining space in a layout. For instance, below is a right-to-left stack of three letters with spacing in between. Between the two spacings, `1fr` takes one third of the remaining space and `2fr` takes two thirds.

```
#stack(
  dir: rtl,
  [A], 1fr, [B], 2fr, [C]
)
```

C B A

A grid layout arranges elements into rows and columns. Each row and column has a defined sizing, which can be either absolute, relative, a mix of the two, fractional, or automatic. Fractional sizing works just like fractional spacing. Automatic sizing measures how much space the largest item in the row or column needs and uses that as the row or column's size. If there is not enough space for all automatic rows/columns, the ones that take up more than their fair share are shrunk proportionally. The grid sizing approach is based on CSS. [58] Note that grids are different from tables. The latter are structural elements that are implemented with grids internally.

Affine Transformations. Typst provides `move`, `scale` and `rotate` functions to affinely transform an element. For example, below we rotate a word by -45 degrees:

```
A #rotate(-45deg)[new] world.
```

A new world.

5.3 Property-Based Styling

The third pillar of the presentational layer is *styling*. We can already arrange building blocks on the pages, but there is no way to configure their appearance. Typesetting is a versatile task. Just for text, there are dozens of properties a user might wish to configure: Font family, size, color, kerning, ligatures, etc. Typst provides *set rules* for this kind of styling (see also Section 3.8).

To understand the reasoning behind set rules, let us start by considering a simple example: Changing the background color of a rectangle. Since Typst revolves around functions, the natural solution is to add a named argument to the `rect` function. Then, the user can write `#rect(fill: red)` to get a red rectangle. This is the approach that many UI toolkits take (e.g., Elm UI and Swift UI). In principle, this approach also easily allows us to apply the same style to multiple elements: We create a new function that binds the `fill` argument to `rect` (*bind* as in partial application). Typst provides the `.with(..)` method to bind arguments to a function. So, to color all rectangles in red, we would redefine the previous `rect` function:

```
#let rect = rect.with(fill: red)
```

This is essentially the *programming-native* solution: It uses the existing programming capabilities instead of requiring a separate styling system. In principle, this is a good solution. However, it comes with a few problems.

Implicit elements. The first problem we face is that not every element we want to style is explicitly constructed through a function call: Text is written directly into the document and list elements have dedicated markup. Still, users want to configure font and list properties. We could tackle this problem by taking the desugaring specified in Section 3.3 even further: Instead of just desugaring enumeration markup to the `enum` function from the standard library, we could desugar it to whatever `enum` function is currently in scope. Styling enumerations could then look similar to styling rectangles. A downside of this fix is that a user might unknowingly call a variable `enum` and break their document.

```
#let enum = enum.with(label: "(a)")                    Does not work!

1. Labelled with (a)
2. Labelled with (b)
```

Composability. The larger problem with implementing styling only in terms of functions is that all styles are then effectively defined by what is in scope. But computational scopes do not compose in the way users expect styling to. This is best illustrated with two examples. Within the functional framework, we would change the main text font like this:

```
#let text = text.with(family: "Helvetica")             Does not work!
This text is in Helvetica.
```

Now, only text that is written at a point where this new `text` function is in scope is affected. As we want to use Helvetica for everything, this functions needs to be in scope in every place where there is text. Within a project, it might be possible to define all styles in the right places so that everything is in scope at the right time. But what if a third-party package would want to provide a `figure` function taking a path to an image and caption text? In the example below, the text “Figure X:” would not be in Helvetica simply because the redefined `text` function is not in scope *within the other package.*

```
// Somewhere in a package.                             Does not work!
#let figure(path, caption) = [
  #image(path, width: 75%) \
  Figure X: #caption.
]

// The main document.
#let text = text.with(family: "Helvetica")
#import figure from "coolfig"
#figure("bird.jpg", [An African Bird])
```

Context. The important insight is that the purely functional approach is doomed to fail because we cannot know the full style of an element at the time it is constructed. The style depends on the context the element is inserted into. This is what set rules enable (see also Section 3.8). The example below shows how the same list containing the same text looks different when affected by different set rules:

```
#let fruit = [
  - Apple
  - Pear
]

#set list(label: [>])
#fruit

#set list(label: [+])
#set text(style: "italic")
#fruit
```

> Apple
> Pear

+ *Apple*
+ *Pear*

With set rules, users can define style properties that affect *all* instances of an element within a block of content, even deeply nested ones and ones that were constructed before the evaluation of the set rule. Still, all settable properties can also be passed directly to a constructor of an individual instance to affect just that instance—meaning that `#rect(fill: red)` still works as expected. In addition, set rules can also style markup. Just like our previous approach, they use the desugaring defined in Section 3.3—but this time without breaking the document if a user decides to name their variable `enum`. The possibility to style elements after their construction is crucial for composability as it enables functions to affect the look of content passed to them, as seen in the last example of Section 3.8.

The capability to have settable properties is what distinguishes functions living in the presentational layer from the ones living in the computational layer. Since styles cannot be resolved without context, set rules are *not just partial application*. Consequently, set rules cannot be used with computational functions and the following does *not* work:

```
#let next(x, skip: 1) = x + skip                              Does not work!
#set next(skip: 4)
```

Chapter 6

Structural Layer

The documents we write are full of recurring structural elements: Headings, sections, lists, figures, and more. With just the computational and presentational layers, Typst users can already create most documents they envision. But these layers are ignorant to the idea of structure. Typst's structural layer lets users express the logical structure of their documents. Through it, Typst markup becomes *descriptive*. And it turns out that this has many benefits.

First of all, structural elements like headings and figures typically have a consistent style. Thanks to the structural layer, users can define the look of an element *once* with a show rule and have Typst automatically apply it to every occurrence of that element. This way, a user can also easily change the look of a kind of element later without manually updating each occurrence. By combining multiple set and show rules into a function, users can create templates that combine a layout with a rich set of styles. At the computational level, templates are simply functions that take a document's full content plus optional configuration and realize the document in a style. This means that, for example, a scientist could switch their paper's style from one conference's to another's just by updating the single line of code that imports the template.

Secondly, there are many cases where some part of a document depends on the remaining document. Prominent examples are the table of contents which depends on the sections; running headers and footers depending on the current section; and a bibliography that lists the works cited in the document. Typst's *introspection* system enables users to express such dependencies. It gives them access to the exact positions where all structural elements ended up in the finished layout. That way, they can easily find all citation elements to build a bibliography or all section headings to build a table of contents with page numbers.

Last but not least, structural information is even valuable in the final output. PDFs, for example, can include structural information to make them more accessible to visually or otherwise impaired people. While our current compiler does not implement this yet, the necessary information is available.

6.1 Structural Elements

An element is structural if it is defined by meaning instead of visual representation. Headings, figures, tables, and footnotes are structural elements whereas rectangles, images, and grid layouts are presentational elements. Similarly, `_emphasized text_` is structural whereas italic text is the specific presentation typically used for emphasis. In Typst, structural elements come into existence in three different ways: Through a function call (like `#heading[Introduction]`), through markup (as in `= Introduction`), or by being built up automatically. For example, text and inline elements automatically build up into paragraphs. Similarly, individual enumeration items build up into an enumeration. At the markup level, there are only items. Typst automatically combines neighboring items into enumerations, as the following example shows:

```
#set enum(label: "1)")
#for letter in "ABC" [
  . #letter
]
```

1) A
2) B
3) C

Some structural elements are invisible: For example, Typst automatically builds up section elements based on headings. While there is no special visual representation for sections, knowing about sections allows Typst to format running headers and to put endnotes in the right places.

6.2 Transformational Styling

A structural element is by definition not tied to a specific presentation—it is a conceptual entity and there may be many different ways to present it. The presentational layer's styling system already allows us to style elements with properties (as we have seen in Section 5.3). However, this kind of styling is restricted to changing predefined options, like how to label enumerations, and it is infeasible for Typst to provide options for everything a user might wish to change.

Instead, Typst's structural layer extends the styling system with *show rules:* These let users completely redefine the presentational representation of a structural element (see also Section 3.9). Below, we have a show rule that displays lists inline with commas instead of block-level with bullet points. Here, `element` is a binding holding an individual list element and the code block after the `as` keyword defines the presentation of that list. The list element's fields give access to both the inherent data of the list (the array of `items`) and the values of all settable properties on `list` (e.g., `label` and `indent`).

```
#show element: list as {
  element.items.join(", ")
}

- List
- With
- Commas
```

List, With, Commas

Each structural element has a default rule, which realizes the element when there is no user-defined show rule. For lists, it is a grid layout with as many rows as items and two columns: one containing the list labels and the other the items' content. Settable properties like `label` affect the behaviour of this default realization. Whether they also affect a user-defined show rule depends on whether the definition uses them: In the inline-with-commas list above, the show rule ignores the `label` property on `element` as it is not meaningful anymore. Thus, setting it would have no effect. In other cases, where the label has a meaning, the show rule could access it through `element.label`.

As mentioned before, Typst automatically builds up certain structural elements, including lists. This build-up interacts with show rules: Structure fragments resulting from a show rule can build up into other elements and built-up elements can be realized with show rules. The example on the next page shows strong elements that realize as list items. These build up into a list, which is then shown separated with commas.

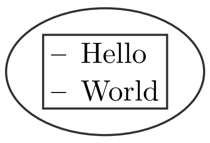
```
#show it: strong as [- {it.body}]
#show it: list as {
  it.items.join(", ")
}

*A* *few* *strong* *words.*
```

A, few, strong, words.

Show rules only work with structural elements, not with presentational ones. This is motivated by the principle of encapsulation: An image could be semantically meaningful or a purely presentational background. Whether an abstraction internally uses an image or something else to achieve a visual effect is an implementation detail. If a show rule could change the look of all images, those abstractions would be less robust. In contrast, structural elements carry meaning by definition and thus cannot be purely presentational.

Recursive show rules. Sometimes we do not want to replace the built-in realization of a structural element but extend it. In Section 3.9, we have already seen the example of placing a rectangle around source code using a single recursive show rule. When there are multiple recursive show rules, the principles remain the same but things get a little more complicated. Typst first executes the show rule that is closest to the element. Only if this rule is recursive, the other rules come into play. Consider the following example:

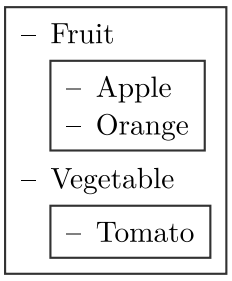
```
#show it: list as rect(it)
#show it: list as ellipse(it)
- Hello
- World
```



In this example, the closest rule is the recursive ellipse rule. Thus, Typst executes this rule upon encountering the list for the first time. The result is an ellipse directly containing the list. Upon layouting the ellipse's content, Typst again meets the list. However, the list has now been *marked* as stemming from the ellipse rule and thus Typst skips the ellipse rule this time, instead using the rectangle rule to realize the list. Without this skipping, every recursive show rule would be infinitely recursive. Upon layouting the rectangle, Typst again sees the list, now being marked as stemming from both rules. Thus, Typst skips both of them and uses the default rule for lists. This rule is not recursive and the list is thus finally fully realized. (If the default rule was recursive, Typst would run out of rules and discard the element.)

Marking. Once an element has been realized with a show rule, it is marked as stemming from that rule. This way, Typst will not consider the rule again if the list is used recursively. The details of this marking are somewhat intricate though. We want to skip the rule for the list itself and all new lists generated by the rule. However, we still want to use the rule for existing nested lists. Consider the following example:

```
#show element: list as rect(
  inset: 6pt,
  element,
)

- Fruit
  - Apple
  - Orange
- Vegetable
  - Tomato
```



– Fruit
  – Apple
  – Orange
– Vegetable
  – Tomato

The top-level list is recursively realized with the show rule. It should not be shown with the rule again. If the show rule would create more lists, those should also not be shown with the rule. Permitting either of these two things could lead to infinite recursion. But we still want the nested lists to use the rule as these are completely independent. Effectively, the rule should be blocked for the whole content coming out of the rule but unblocked for all content going into the rule (except for the main element). This way, the rule will neither affect the element itself nor any synthetic elements created by it, but nested elements can use the rule again as expected. Figure 2 shows which parts of the content in the example above would be blocked and unblocked after the first application of the show rule.

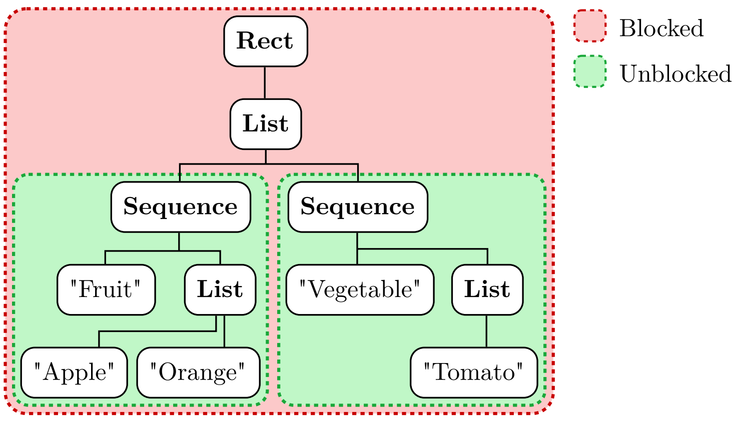


Figure 2: Blocking and unblocking for the example above. The red part of the tree is blocked from using the list show rule again. The two lists within the green parts can and will use the show rule again. In the end, both will be wrapped in rectangles, too.

Text styling. The final ingredient of the extended styling system are text patterns. These allow users to replace a specific string or any text matching a regular expression with arbitrary content. Especially by mixing regular expression patterns with local show rules, users and template creators have enormous freedom in creating automations and simplifying their markup. On the next page, we use such a show rule to write a function that automatically transforms all numbers in its body into Roman notation. Just as before, Typst marks rules as blocked to prevent infinite recursion. However, with text rules, there is no need for unblocking as text cannot contain nested text.

```
#let romanized(body) = {
  show v: regex("\d+") as {
    roman(int(v))
  }
  body
}

1, 2 and 3 are normal. \
#romanized[1, 2 and 3 changed.]
```

1, 2 and 3 are normal.
I, II and III changed.

6.3 Introspection

Documents are intradependent: Some parts of documents depend on other parts. The table of contents depends on the sections and headings in the document. A running header similarly depends on the current section. A bibliography at the end of a paper or thesis depends on which works were cited within the document. And last but not least, within a document there are often cross-references to other sections, figures, or tables. Maintaining all these intradependencies manually is error-prone and laborious.

We present a principled and automatic approach to resolve such intradependencies called *introspection.* It enables users to locate elements within their document in two different ways: First, by *kind*, for example to find all code blocks, figures, or tables within a document. Second, by *label*, to find a specifically labelled element. Typst provides not only the discovered elements but also the page numbers they are on and even their exact positions. Since users can create arbitrary content from this information, they have the flexibility to completely customize their table of contents, to create custom indices, and much more.

Intra-document dependencies are cyclic in nature. For instance, a heading might originally have been on page 5, but once added to the table of contents, the table of contents breaks across two pages and the heading is suddenly on page 6. These dependencies pose problems for typesetting software. Microsoft Word and TeX show different symptoms of the same problem: In Word, users have to manually update their table of contents through a click of a button. Meanwhile in TeX, they need to compile multiple times for changed headings to propagate to the table of contents. Fundamentally, Typst faces the same problem as TeX. However, Typst is built with introspection in mind and automatically relayouts the document as many times as necessary to resolve all dependencies.

Locating. To express intra-document dependencies, Typst provides a mechanism to access the document's structure and the locations of structural elements in the final layout. This mechanism is the `locate` function. It takes two arguments: First, a structural element or a label to search for; and second, a function that maps an array of discovered elements to displayable content. It returns opaque content that resolves to the return value of the passed function. With the `locate` function, we can easily build a list of figures as follows:

```
#locate(figure, list => {
  for i, fig in list [
    Fig. {i+1}: {fig.caption}
    #repeat[.] {fig.page}
  ]
})

...
```

```
Fig. 1: The Earth ............................. 2
Fig. 2: The Universe ......................... 3
Fig. 3: The Meaning of Life .............. 5
```

Consider an alternative, seemingly simpler design in which the `locate` function directly returns the located elements instead of providing them through a callback: In this design, `locate` would not be a pure function as its return value would depend on the remaining document. In contrast, with the callback approach the `locate` function *is* pure. The principle is similar to how Haskell enables I/O with pure functions: The opaque content value returned by a `locate` call is completely independent of which elements will later be found. It is always the same for the same combination of element/label and function. Only during layout, this content is interpreted, just like Haskell's runtime interprets the I/O action defined as `main`. [59]

Note that `locate` only works with structural elements, not with presentational ones, for the same reason that show rules only work for structural elements: Structural elements have a role and meaning within the document. In contrast, it is an implementation detail how exactly a component uses presentational elements. Support for locating arbitrary presentational elements would break the principle of encapsulation. However, there is an *opt-in* way to make any element locatable: Annotating it with a label. This is useful both for locating a specific structural element (like a specific section heading) and for locating a presentational element to achieve a custom effect. The example below illustrates how we can label multiple presentational elements and then find them with `locate`. Since there can be multiple elements with the same label, `locate` still provides a list of elements.

```
Number of favorite elements:
#locate("fav",
  list => list.len())

#circle(height: 10pt) <fav>
#square(height: 10pt) <fav>
#rect(height: 10pt)
```

Number of favorite elements: 2

This is also how cross-references are implemented. A reference uses `locate` to find a unique element with the label, inspects what kind of element it is (heading, figure, etc.), determines its index among the other elements of that kind (through a second `locate` call), and formats the result humanly readable (e.g., "Section 5" or "Figure 3"). If there are zero or multiple elements with the label, the reference results in an error. A reference is itself a structural element and can be customized with a show rule.

So far, we have only seen how to locate *all* elements of a specific kind or with a specific label. But what if we wanted to find out where a *specific* element is in relation to the other elements of its kind, for instance to number the section headings of a document?

While a show rule containing a `locate` call can find all headings, we have no way to identify the specific heading currently being shown among them. To support this use case, the function passed to `locate` can optionally have two additional parameters, `before` and `after`. These indicate the position of the `list` items in the document flow relative to the content returned by the `locate` call. The first specifies how many items are logically before the locate call, and the second how many are after it. In the example below, we number section headings using `locate`. The number of each heading should be one plus the number of headings before it (since the numbering should start at one). This directly translates to the following show rule:

```
#show it: heading as locate(
  heading,
  (list, before, after) => [
    {1 + before}. {it.body}
  ]
)

= Alpha
= Beta
= Gamma
```

1. Alpha

2. Beta

3. Gamma

Resolving. Resolving introspections is an iterative process. Initially, Typst has no knowledge of the position of any element and thus has no information it can provide to introspections. In a document without introspections, this is unproblematic. If there are introspections though, Typst has to layout over the course of *multiple rounds.* In the first round, it determines initial positions of structural and labelled elements. In the second round, it can provide this information to the introspections. These produce content, which in turn can affect the layout and thereby the positions of other elements. If the positions after the second round differ from the ones after the first round, a third round of layout becomes necessary. This process repeats until a fixpoint on the positions is reached. See Section 7.5 for details on how the Typst compiler tracks the positions of elements.

A change to the positions does not always necessitate a relayout though. The significant question is whether the change is *observable.* If all `locate` queries made on the old positions yield the same result on the new ones, the observed part of the input stays unchanged. As Typst is deterministic, a relayout would produce the same output and is thus unnecessary. For example, a heading might slightly change its position during a layout round, but the only locate query for headings is merely interested in the total number of headings and not their individual positions. Comparing only the observed input is an effective optimization as most of the time not all available information (e.g., exact positions) is actually inspected.

Some introspections are fundamentally unstable. In the example on the next page, we locate all headings, determine the page number p of the first heading, and then insert p page breaks before that heading. In the initial layout round, the `locate` callback is skipped and the heading is placed onto the first page. Since there was a `locate` call, layout starts once more. This time, the `locate` callback is executed, locating the heading on page one. Consequently, the code produces a page break and the heading drops down to page two. As the position is different from before, a third layout round starts, in which the heading moves to page three. On each relayout, the document grows by one page. The result never stabilizes.

```
#locate(heading, list => {
  let p = list(0).page
  p * pagebreak()
})

= Introduction
```

Even worse, in general, Typst cannot know whether an introspection will stabilize. Consider the generalized case of a single element and a single locate call. This call determines the page the element is on and, through page breaks, decides on which page the element will be in the next iteration. Mathematically speaking, the position of the heading is defined through a recurrence relation a where $a_1 := 1$ and $a_n := f(a_{n-1})$. Here, $f : \mathbb{N} \to \mathbb{N}$ is defined through the code in the function passed to `locate`. The introspection is stable if and only if a converges to a fixed value. Thus, to decide whether an introspection is stable, Typst would need to be able to prove that a converges for an arbitrary computable function f. This is a *non-trivial*[2] behavioural property of f: It holds true for $f(n) = 1$ but not for $f(n) = n + 1$. Thus, due to Rice's theorem, it is undecidable for an arbitrary f. Typst cannot analyze whether an arbitrary introspection stabilizes. Luckily, typical introspections stabilize within two to three passes. Thus, a Typst compiler can simply stop compiling after a small constant number of tries, producing a user-facing error message at the guilty `locate` call.

[2] In this context, *non-trivial* means that the semantic property is true for at least one function but not for every function.

Chapter 7

Compiler

We have implemented a compiler for the Typst language. It transforms a collection of Typst files into output formats like PDF or PNG. At the time of writing, it fully implements the Typst language as specified in this thesis except for certain parts of the structural layer and math typesetting. These areas are still in active development. The most significant missing pieces are labels, references, and the `locate` function as discussed in Section 6.3. The compiler does, however, implement a previous version of the introspection system that had a different user interface. This earlier system was used for section and figure numbering, the table of contents, cross-referencing, and citations in this thesis.

Users of modern programming languages expect more than just a batch compiler—from syntax highlighting to autocompletion there are a lot of supplemental tasks. For many languages, these are separately developed as part of *language servers* that integrate with code editors. With Typst, we follow a different approach: While the compiler has a standard command line interface, it can also be consumed as a library (specifically, a Rust crate). This library implements both the main transformation from source files to output formats and additional tasks like syntax highlighting. It loads fonts and files through an abstract interface so that it can be ported to many different environments.

Like many compilers, the Typst compiler has several *intermediate representations* that lie in between the source text and the output formats:

1. *Syntax Tree:* Represents the syntactic structure of a file. The tree is lossless (spaces, comments, etc. are retained) and not strongly typed (any kind of node can have any other kind of node as a child). This way, the tree can be traversed very easily, making it suitable for tasks like syntax highlighting. The *evaluation* operates over a typed view on top of the syntax tree. This view enforces the correct syntactic structure.

2. *Content Model:* Tree data structure underlying the content data type. Represents text, spacing, layouts, graphical elements, and so on. The model is partially styled, that is, it can contain style information inside but is also reactive to "inherited" styles from the outside.

3. *Frames:* Finished layouts of fixed sizes that contain primitive elements, affinely transformed subframes, and *locators* at fixed positions. The locators are used to resolve introspection during *lifting.* Each output page is represented by one top-level frame.

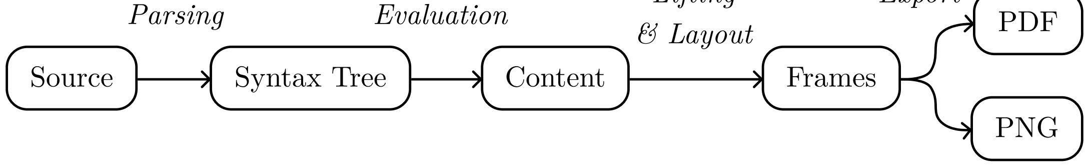


Figure 3: Representations and transformations.

Five transformations convert between the input, the intermediate representations, and the output as illustrated by Figure 3:

1. *Parsing:* Transforms a string of source text into a syntax tree.
2. *Evaluation:* Transforms a parsed source file plus other source files and assets referenced by it into content. In particular, this executes embedded code blocks and function calls. Alongside the content, evaluating a file produces a scope of top-level bindings which another module can import.
3. *Lifting:* Transforms the flexible content model into directly layoutable nodes such as flows and paragraphs. Content may contain directly layoutable nodes, but more often than not it consists of higher-level elements like lists, headings, or paragraph breaks.
4. *Layout:* Transforms a layoutable node into a sequence of frames. Happens intertwined with lifting: If layouting encounters content that is not directly layoutable, it first lifts it. Multiple rounds of layouting may be necessary to stabilize all introspections.
5. *Export:* Transforms frames into output formats like PDF or PNG. This separation makes it easy to add support for more output formats or direct preview rendering.

Note that these are specifically called "transformations" and not "phases" or "passes". The compiler does not execute them strictly in order. For example, evaluation can trigger parsing when a module is imported and lifting can trigger evaluation and by extension even parsing when executing show rules. Even more importantly, lifting and layout are deeply intertwined. Whenever layout encounters high-level content, it triggers lifting. Lifting, in turn, is a shallow operation: Lifted content might still contain non-layoutable content nested within layoutable nodes.

7.1 Parsing

The Typst compiler uses a hand-written recursive descent parser to transform an input string into a syntax tree. An approximate EBNF grammar derived from this parser is attached in Appendix A. Because EBNF grammars can only describe context-free languages [60], this grammar does not handle the indentation rules for list markup.

Syntax errors are never fatal, they simply result in the syntax tree containing error nodes. If a tree contains any such error nodes, it is not well-formed and the compiler does not attempt to evaluate it. However, the tree can still be used for syntax highlighting or similar tasks as errors are mostly local.

Typst's parser is *incremental:* For local changes, it reparses only a part of the source file. For more details on this incremental parsing scheme, see M. Haug's master's thesis. [8]

7.2 Syntax Tree

Typst's syntax tree represents a full Typst file. It is suitable as the input to evaluation as well as for supplemental tasks like syntax highlighting and autocompletion. Each node in the tree has a unique number attached to it (its *span number*) that is stable across multiple compilations. This allows later stages of the compiler to refer back to a specific syntax node (e.g., for error reporting) without hurting cache performance.

The syntax tree is *lossless,* that is, every piece of source text has a corresponding node (even whitespace and comments). This makes it suitable for syntax highlighting. Typically, language syntaxes for highlighting are defined using regular expressions. However, Typst's syntax is quite difficult to process with regular expressions because determining the end of a hashtag expression requires knowledge of the full grammar. Implementing syntax highlighting directly on the syntax tree has the added benefit that changes to the language's syntax need only be made in one place. Figure 4 shows an example of a lossless Typst syntax tree.

A classical AST (abstract syntax tree) implementation defines concrete types for all syntactical constructs. A problem with ASTs is that routines operating only on parts of it must in many cases still know how to traverse the remaining tree. A typical solution to this is the *visitor pattern* which implements an abstract traversal that can be used in different places of the compiler. [61] Typst's syntax tree is untyped, that is, the tree representation allows every kind of node to have every other kind of node as a child. This fixes the aforementioned problem, as the tree is trivially traversed with a few lines of recursive code. Further benefits of untyped syntax trees are that it is simpler to make them lossless and that they can even represent partially malformed input.

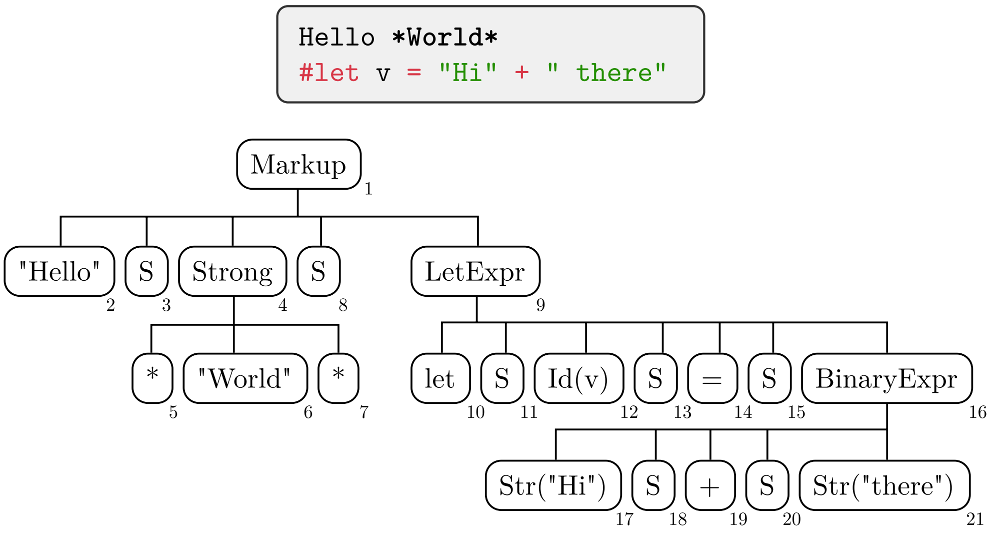
In the end, the evaluation still expects a specific syntactical structure though. For this reason, Typst provides a typed layer on top of the untyped syntax tree. This layer consists of more classical AST types that each wrap a syntax node and provide accessors for their constituent parts. Should the syntax tree not conform to the expected syntactic structure (i.e., if there is a bug in the parser), the typed layer catches the error and fails at runtime.

If an error occurs during layout, the compiler reports it back to the user. To make the error instructive, it keeps track of where in the source code a layout-phase construct originated from. A simple way to keep track of a node's source location is through its byte offset in the source string. However, upon an insertion, the byte offsets of all nodes after the insertion change. If byte offsets were part of the input to later stages, cache performance would be severely hurt (see Chapter 8). Instead, Typst implements *numbered spans:* These are unique IDs assigned to each syntax tree node that stay stable if another part of the file is incrementally reparsed. The numbered spans in a tree adhere to a strict ordering, enabling the compiler to efficiently find the node corresponding to a given span number.

7.3 Evaluation

The evaluation phase transforms a source file into a module containing content and top-level bindings. It is implemented in the form of a *tree-walking interpreter.* Such an interpreter simply walks over the source file's syntax tree and evaluates each node. At the top level, each file consists of markup. The individual markup nodes evaluate to content values, which the interpreter joins together. When encountering an embedded code expression, the interpreter first evaluates it to a computational value and then displays said value as content.

During the course of evaluation, the interpreter manages a stack of scopes, each being a map from identifiers to values. When entering a code or content block, it first pushes a new, empty scope onto the stack. Then, it evaluates step by step each expression or



```
Hello *World*
#let v = "Hi" + " there"
```

Figure 4: Example source code and its lossless syntax tree representation. The "S" nodes represent spaces in the input and the small numbers are the span numbers.

markup node in the block, joining the results in the process. Before exiting the block, it removes the topmost scope from the stack and discards it. When encountering a let binding, the compiler simply inserts the binding's name and value into the topmost scope on the stack. That way, it will live only until the end of the surrounding block.

In the example below, the interpreter starts by evaluating the let expression. To do so, it evaluates the contained binary `+` operation, simply by recursively evaluating the subtrees of both sides and then summing up the two resulting string values. The let expression yields a `none` value but has the side effect of writing the string `"Hello"` into the topmost scope. Displaying a none value produces no content at all. Next up is the `#hello` expression. It has no side effects and yields the string `"Hello"`. Displaying this string value produces text, which is visible to the right.

```
#let hello = "Hel" + "lo"
#hello
```

Hello

As discussed in Section 4.3, the `break`, `continue`, and `return` control flow primitives work somewhat differently in Typst: They do not stop the execution of a loop or function body instantly. Rather, they only stop the evaluation of all blocks between them and the loop or function early. This approach is quite simple to implement: The interpreter just stores an optional current control flow event. Then, after each expression in a code block or node in a content block, the interpreter checks whether there is a current flow event and if so, stops further evaluation of the block. Similarly, the interpreter checks for a control flow event after each iteration of a loop and after the execution of a function's body.

To evaluate a closure, the compiler searches the body of the closure for variables that are *accessed* in the closure but not *defined* in it. It then reads those variables from the scope stack and stores them alongside the closure's body in a closure value. Later, when the closure is executed, the interpreter creates a clean scope stack and stores in it read-only copies of all captured variables. The captured variables may not be written to because that could make the function impure:

```
error: cannot modify captured variable                    DIAGNOSTIC
   ┌─ main.typ:2:11
   │
 1 │ #let count = 0
 2 │ #let f() = count += 1
   │            ^^^^^
```

To evaluate an import or include expression, the interpreter evaluates the path expression, loads and parses the source file at the path, and then tries to recursively evaluate the source file. Even when imported multiple times from different locations, the compiler evaluates each module only once. Further imports (or includes) of the same module take the finished module from the cache. At all times, the interpreter keeps track of the *route* of modules it is currently recursively evaluating. If an import path points to a file that is already part of the current route, the compiler rejects the import as cyclic and produces an error message:

```
error: cyclic import                                      DIAGNOSTIC
   ┌─ text/main.typ:1:9
   │
 1 │ #include "main.typ"
   │          ^^^^^^^^^^
```

Set and show rules are only partly processed during evaluation. For set rules, the compiler evaluates the arguments and produces a *style map* containing the properties (see Section 7.4). For show rules, it creates a function value for the mapping from structure to presentation and stores that in a style map, too. The style maps only come into play later, during lifting and layout.

7.4 Content Model

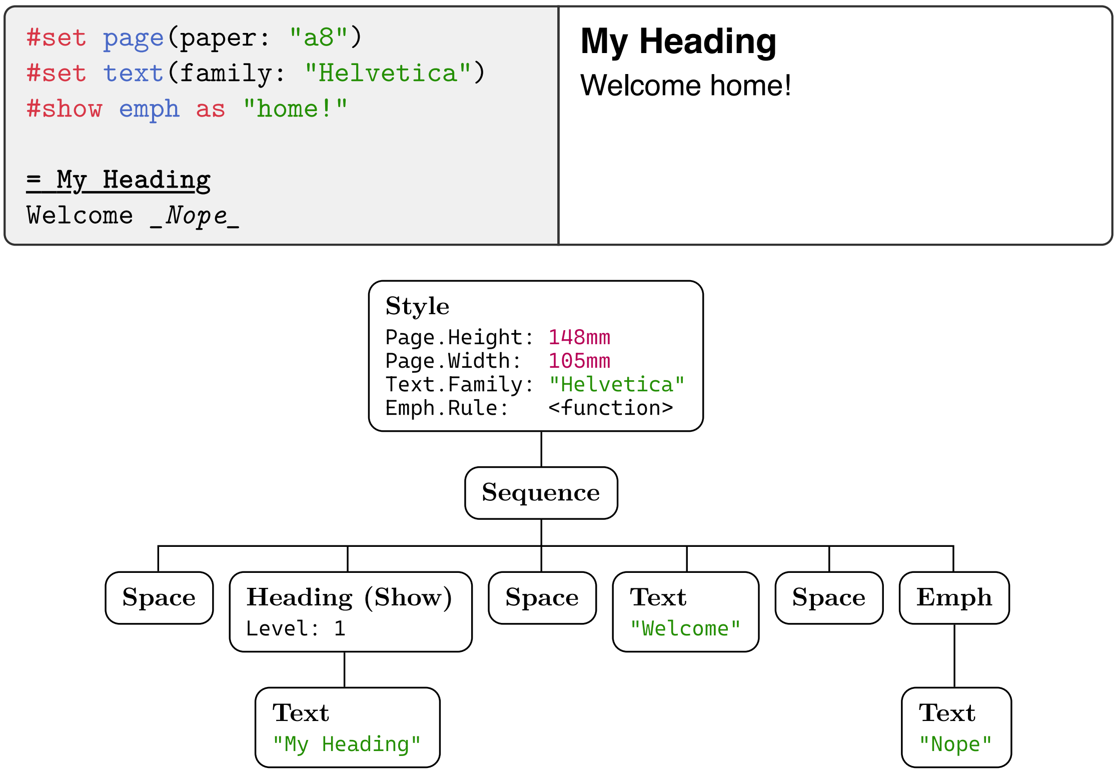
The content model encodes a mix of structural elements, presentational elements, and styling. It is a tree with different kinds of nodes: The tree's leaves are basic building blocks like text or spacing. Its inner nodes include structural and presentational containers as well as sequence and styling nodes. The nodes fall into six categories:

- *Basic Leaves:* Text, text spaces, fixed spacing, paragraph breaks, and more.
- *Block/Inline Nodes:* Directly layoutable nodes. Block nodes become part of flows during lifting while inline nodes become part of paragraphs.
- *Structural Nodes:* Hold arbitrary structural elements that the compiler maps to their presentational representations during lifting.
- *Locate Nodes:* Nodes that (a) encapsulate an introspection and (b) can be located through other introspections. In the current compiler revision, introspections can only locate these specific nodes, not arbitrary structural or labelled elements. This is not a fundamental restriction as the show rule for a structural element can generate a locate node.

- *Style Nodes:* Hold a *style map* and a nested node. The style map contains style properties from set rules and recipes from show rules that apply to everything contained in the nested node. The compiler chains style maps together to *style chains* during lifting.
- *Sequence Nodes:* Hold a list of nested nodes. Joining content in Typst results in this node: `[a] + [b]` creates the node `Sequence(Text("a"), Text("b"))`. Sequences may be nested transparently; different sequence trees that flatten to the same result are equivalent.

A node's style is affected by all style nodes between it and the root node. During layout and lifting, this style information is captured in the *style chain.* This data structure efficiently combines all style maps up the tree without any copies or allocation. It works similarly to a linked list. A style chain link holds two pointers: One to a style map and one to an outer chain link. To find a style property's value, the compiler checks the innermost link's style map and, if that one does not contain the property, it moves up to the next link. If there is no entry in any chain link, the property takes on its default value.

Example. Figure 5 presents an example of markup and its corresponding content model. As the markup starts with set and show rules, the model's root is a *style node* containing a *style map.* This map contains the set rule properties and show rule recipe. The `paper` property is already resolved to `width` and `height` properties at this point. The show rule is stored as a function, which will be called when an emphasis node is encountered during lifting. Moving on, the tree contains two structural nodes, a heading and an emphasis node. The former becomes larger and bold through the default recipe



```
#set page(paper: "a8")
#set text(family: "Helvetica")
#show emph as "home!"

= My Heading
Welcome _Nope_
```

Figure 5: Example markup and its content model.

while the latter is replaced by the text "home!" due to the show rule. In between and around all the nodes are space nodes. During lifting, the compiler removes duplicate and outer spaces, so that the only remaining one is between "Welcome" and "home!"

7.5 Lifting

To layout content, the compiler first turns the flexible free-form content representation into a block-level *flow node* or a top-level *document node.* We call this process *lifting.* During lifting, many interesting things are happening, in particular show rule application and introspection.

To build a flow (or document) node from content, lifting initializes a context of *builders* into which the content is integrated step by step. Currently there are four builders: A list, paragraph, flow, and document builder. (The last one only exists when lifting to the top-level document.) When a node arrives at the builders, each of the builders, in order, gets a chance to accept it. If it accepts the node, the node becomes part of the structure the builder is constructing. If not, the builder is *interrupted* and, before the compiler further processes the node, the builder finishes up and lifts its own result. For example, the list builder accepts list items but would be interrupted by a text node. A paragraph, on the other hand, accepts text, spaces, and inline nodes but would be interrupted by a grid. In the end, once content has been lifted, the compiler once more interrupts all builders, to finish up any content accumulated within them.

Now, to lift nodes from the six categories discussed in Section 7.4, the compiler proceeds as follows:

- *Basic Leaves:* Except for text nodes, these directly go to the builders. Text nodes require special treatment because they can be affected by text show rules. To lift a text node:

  1. The compiler traverses the style chain in search for a matching text or regex rule. If it finds a rule that is not already marked as used, it applies that rule to the text, producing replacement content.
  2. Next, it marks the replacement content as stemming from the rule. A marker identifies a show rule through its position from the top of the style chain. This works because the replacement is layouted with the same style chain as the original structural node.
  3. Finally, the compiler lifts the marked replacement. During this lifting, further styles might be added to the bottom of the chain, but the trunk of the chain stays the same. (Which is why the marker uses the position from the top, not the bottom.)

  If multiple rules match the text, the compiler still only applies the first rule to the text. However, if the replacement still contains the matching text, the other patterns get a new chance to match during lifting of the replacement. If there is no matching pattern, the text moves on to the builders.

- *Block/Inline Nodes:* These are directly layoutable and thus they directly go to the builders.

– *Structural Nodes:* Similarly to text nodes, for structural nodes the compiler traverses the style chain, searching for a matching rule. If it finds a matching one, it applies that rule to the node, marks the replacement, and recursively lifts it. If there is no matching rule, it executes the base rule for the node instead.

– *Locate Nodes:* Over the course of multiple layout rounds, the compiler retains a locator context. Within this context, it stores the positions of all locate nodes and a counting index n. Before the first layout round, the context is empty and $n = 0$. To lift a locate node, the compiler does two things:

  1. It generates a locator for n and sends it to the builders. This locator will make its way into the final frames, where the compiler can find it to update the position of locate node n for the next round.

  2. It uses the last round's context to compute the result of the user's introspection and passes that result to the user's function. The function returns content, which the compiler once again lifts.

  After lifting the node, the compiler increases n by one, and at the start of a new layout round it resets n to zero. During each round of layouting, the context can change and grow. If the positions stay stable and unchanged for one whole round, no more rounds are necessary.

– *Style Nodes:* To lift a style node, the compiler appends the node's style map to the current style chain and and lifts the node's child with the new extended style chain.

– *Sequence Nodes:* A sequence node is lifted simply by iterating over all its children and lifting them individually.

Example. Lifting is best illustrated with an example: Consider the markup and content depicted in Figure 6. To lift this content, the compiler first lifts the style node, hooking up the show rule into the style chain. Then, it proceeds to lift the sequence node, recursing to lift the children:

1. The first child is `- One`. Because it is a list item, the list builder accepts it.

2. Next up is `- Two`, which the list builder once again accepts.

3. What follows is the text node `Interrupt`. There is no text recipe, so the text directly goes to the builders. The list builder does *not* accept text and is therefore interrupted. The compiler creates a new list builder, finishes the old one into a list node (which is a structural node), and lifts that node recursively. As there is a show rule for lists, this results in the text node `Hello`. This new text node is promptly rejected by the new empty list builder. Although that list builder technically gets interrupted, it is empty and thus nothing happens. The text moves on to the paragraph builder, which accepts it. Only now, the compiler returns to the original text node `Interrupt`, which is too accepted by the paragraph builder.

4. Finally, the compiler reaches the last node in the sequence: `- Three`. As it is a list item, the new, empty list builder accepts it. Now all nodes are lifted and the compiler performs one last step: It interrupts all builders, and in that way, our final list becomes another `Hello`.

```
#show it: list as [Hello]
- One
- Two
Interrupt
- Three
```

```
Hello Interrupt Hello
```

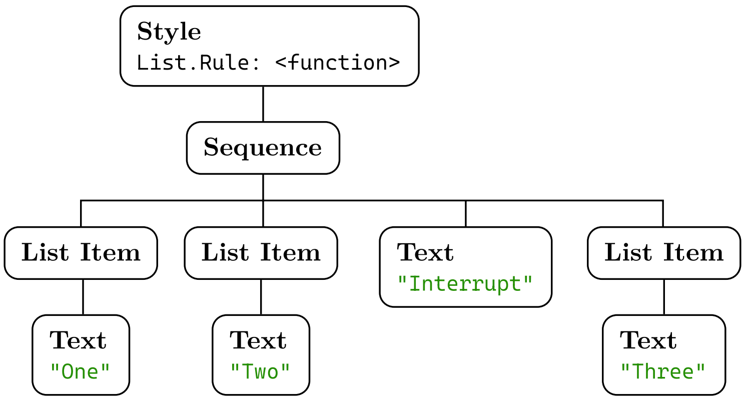


Figure 6: Example of markup and content model to illustrate lifting. Space nodes between the list items and text nodes are omitted for clarity.

7.6 Layout

The layout phase transforms layout nodes into exportable frames. All these nodes (flows, paragraphs, stacks, grids, etc.) implement a common *layout interface:* Layout takes a node, a sequence of *regions,* and a *style chain,* and produces a sequence of *frames.* The region sequence defines fixed-sized areas into which the node's content is layouted. Its length is at least one and potentially infinite. The top-level region sequence, for example, consists of an infinite amount of page-sized regions.

The region sequence constraints the frame sequence. The latter's length is also at least one but at most the length of the region sequence. Similarly, no frame in the frame sequence may be larger than its corresponding region. Furthermore, a region can specify that the node should *expand* along the X, Y, or both axes, in which case the frame's width, height, or both must exactly match the region's size. This is, for instance, used to ensure that the top-level paragraphs have exactly the width of the page minus the margins. The final feature of regions is that they can specify a *base* size in addition to their natural size. This size is used for relative sizing.

Multiple rounds of layout might be necessary to stabilize the positions of all locators. At the start of each round, the compiler compares the current locator context to the previous's round context. See Section 6.3 and Section 7.5 for more details.

Example. Let us look at what happens during layout of the top-level flow of paragraphs. This flow shall be layouted into an infinite sequence of regions with size of the inner page area and expansion enabled for both axes. If expansion was disabled for one of the axes, layout would not produce a page of the correct size.

Before starting, the flow layouter disables expansion along the Y axis for its children so that the first child does not simply take up all the space. Then, it layouts its first child, in this example a paragraph of a few dozen lines that takes up 30% of the first page. Reflecting this, the flow layouter adjusts the first region's remaining height to only 70% of the original height. Then, it moves on the next child, an image whose height is specified at 40%. The user, of course, intended for the image to be sized at 40% of the full inner page height and not 40% of the remaining 70%. This is why it is important that the regions also stores their base (unmodified) size. The image results in one frame and now 30% of the first region's space remain (assuming there is no paragraph spacing).

The final paragraph is a bit longer. It does not fit into the remaining 30%, so it results in two frames. The first one fills up the remaining space in the first region and the second one starts eating into the second region. In this way, the flow layouter proceeds, updating regions and layouting children until no more children are left.

7.7 Frames

Layouting content results in *frames.* A frame is a box of fixed size containing elements at fixed positions. Each element is either a primitive, a linearly transformed subframe, or a locator.

There are currently four kinds of primitives:
1. Shaped text in the form of a glyph run,
2. A geometric shape with fill and stroke,
3. A vector or raster image,
4. An internal link to another page or an external hyperlink.

A locator is an element that is generated by a locate node during lifting. It acts as a beacon that the compiler can find after layout is finished to know where the locate node's content ended up. A locator identifies a locate node through its index among all locate nodes. For more details, see Section 7.5.

7.8 Export

Different exporters transform frames into different output formats. The Typst compiler supports PDF and raster graphic output (PNG).

The PDF exporter creates a PDF file containing content streams for all pages plus embedded fonts and images. It removes unused data from fonts (e.g. outlines of unused glyphs) to reduce the file size. Apart from purely visual content, PDF files can also encode a document's structure. At a language level, Typst has this rich structural information. However, the current implementation does not yet make the best use of it. Since the frames currently only contain locators and not a full reverse mapping to the document's structure, the exporter cannot write this structure information. As a result, PDFs produced by the Typst compiler are not yet accessible for visually impaired people.

The PNG exporter directly renders the frames to a pixel buffer. This is a very straightforward task as the frame representation is ideal for this.

Chapter 8

Evaluation

In the following, we evaluate the Typst language, TeX/LaTeX, and XML/CSS/XSL based on the two criteria Simplicity and Automatability introduced in Chapter 1. There will always be trade-offs to be made between these two criteria. Powerful automation requires abstraction, and abstraction always introduces complexity. Nonetheless, similar automations can strongly vary in how much complexity they incur. We systematically discuss simplicity in terms of syntax, semantics, and diagnostics. Similarly, we examine automatability in terms of foundations and supplemental systems.

There are, of course, other relevant criteria we could discuss, including flexibility and performance. Flexibility, for instance, measures which of all possible documents a solution can produce. This is not particularly interesting, though, as a plain program that only supports affinely transforming glyphs, shapes, and images can produce all the same documents Typst and TeX can—just like modern programming languages are not fundamentally more powerful than Turing machines. Regarding performance: While some markup languages are more amenable to fast compilers than others, performance characteristics are properties of a specific implementation rather than of a language. For an in-depth performance evaluation of the Typst compiler, consult the work of M. Haug. [8]

8.1 Simplicity

Although the overall simplicity of a system is subjective, there are certainly objective factors which make a system simpler or more complex. In the following, we first discuss conciseness and readability of Typst's, TeX's, and XML's syntax. We then move on to semantics, with particular focus on how robustly Typst and TeX behave in tricky situations. Finally, we compare the clarity of Typst's and TeX's error messages. While such messages are (like performance) a feature of a compiler, a language's design strongly influences the ability of a compiler to produce them.

Syntax. Lightweight markup languages like Markdown have grown very popular even though most of them differentiate themselves from HTML only through simpler syntax. This shows how much users value clear and compact syntax in a markup-based authoring system. Compared to Markdown, HTML, XML, and LaTeX are quite verbose. LaTeX environments are the worst offenders in this regard: Their overhead consists of the words "begin" and "end", six special characters and twice the environment's name. An HTML element only requires five special characters and twice the element's name. Meanwhile, Typst functions and TeX commands are about as short as it gets: the function's or command's name plus three special characters. For a direct comparison of these three variants, see example three of Table 3. Notwithstanding the increased verbosity, repeating an element's name at its end could be seen as advantageous since it clarifies which element ends where. By itself, neither lightweight markup nor macro/tag/function-based markup is satisfactory: The former lacks flexibility while the latter is too verbose. For this reason, Typst combines both approaches: Lightweight markup for

Typst	LaTeX	HTML
`_Hello_`	`\emph{Hello}`	`<em>Hello</em>`
`- Apple` `- Orange` `- Banana`	`\begin{itemize}` `\item Apple` `\item Orange` `\item Banana` `\end{itemize}`	`<ul>` `<li>Apple</li>` `<li>Orange</li>` `<li>Banana</li>` `</ul>`
`#quote[` `Time without ...` `]`	`\begin{quote}` `Time without ...` `\end{quote}`	`<blockquote>` `Time without ...` `</blockquote>`
`{1 + 2}`	`\newcount\sum` `\sum=1` `\advance\sum by 2` `\the\sum`	`<script>` `document` `.write(1 + 2)` `</script>`
`#let hi = [Hello]`	`\def\hi{Hello}`	N/A

Table 3: Comparison of Typst, LaTeX, and HTML syntax.

the common cases and minimum-overhead function calls for the remaining ones. Typst's lightweight markup largely borrows from Markdown, as seen in the first two examples. This way, Typst is instantly familiar to the many existing users of Markdown.

As TeX expresses almost everything through macros, it has a very minimalist syntax surface. In theory, this should make it very easy to learn. However, in many cases, the syntax still needs to express complex constructs. Then, TeX code turns into a sea of macros as observable in the fourth example. Typst, on the other hand, has discrete syntax for common structural elements as well as coding constructs. Its syntax uses far more special characters than TeX's and is overall more complex. However, for exactly this reason, Typst code tends to be shorter and more visually structured. This improves readability and allows for better syntax highlighting. Of course, this is only an overall tendency and not a rule: The fifth example shows a case where both the Typst and TeX code are clear and the TeX variant is shorter. In this case, the TeX code might even be clearer for beginners without programming background. HTML and XML are bystanders to this discussion as they do not have built-in programming constructs.

LaTeX has two ways to represent structural elements: Commands like `\emph{...}` and environments like `\begin{itemize}...\end{itemize}`. Both stand on equal footing. Which one to use depends on the specific element in question. Commands typically represent shorter constructs like emphasis or a section heading while environments are mostly used for block-level constructs that themselves contain structure (lists, figures, tables, …) The technical reason is that environments have more control over how macros expand within them, enabling things like the `\item` command for lists. This, however, is an implementation detail that most users should not need to worry about. Still, users will find themselves in situations where they are unsure whether to use a command or an environment.

As discussed before, Typst also has two ways to construct elements: through lightweight markup (`_Hi_`) and through functions (`#emph[Hi]`). However, in Typst, lightweight markup and functions do not stand on equal footing. The former is just syntactic sugar for the latter. This is an important distinction: If a language has multiple ways to do something at the same level of abstraction, that could be an indicator of unnecessary complexity. Meanwhile, having multiple ways to do something at different levels of abstraction is inherent to how abstraction works: It combines lower-level pieces. The problem of feature overlap also manifests itself in the Web's languages: Since each problem has its own domain-specific language, there are many duplications: CSS selectors and XPath solve similar problems, XSL and JS have duplicate programming constructs, and XSL and CSS both support property-based styling.

Coming back to TeX, there are two more notable syntax-related features: First, TeX programs can modify TeX's syntax *during execution* by assigning new *catcodes* (category codes). The catcode of a symbol determines how the parser processes it. By reassigning catcodes, users can change the meaning of pretty much every symbol, including the comment (`%`), grouping (`{}`) and command (`\`) characters. Second, some seemingly syntactical structures are actually not syntactical, at all. Even though a formula such as `$x+y$` is delimited by dollar signs, the formula is not a syntactical entity. Instead, the dollar signs are commands that switch TeX into and out of math mode. This means that a formula can start with `$` and end with `\)` (an alternative delimiter for formulas). While TeX happily accepts this, it confuses any tool that does not perform full macro expansion (e.g., syntax highlighters). Typst and XML do not have this problem as they have well-defined syntaxes.

Semantics. Familiarity matters for semantics just as much as for syntax. Although macros have been around for a long time, in programming, they are mostly used for code generation, not for logic. Programming with functions, loops, and other imperative concepts is much more familiar to most programmers than with macros—making Typst easier to use for algorithmic tasks. In some cases, Typst adapts the semantics of common constructs. The prime example for this is *joining* in blocks and loops. When coming from other programming languages, it is initially confusing that a block is not only an expression but one that combines everything inside of it. In spite of that, joining simplifies many typical tasks and increases the overall consistency of the language. Through joining, code and content blocks are conceptually closer and one can go from one to the other by performing a *block flip* (see Section 4.3).

Compared to TeX, an important benefit of Typst is that fewer things can go wrong. Since there are no package conflicts thanks to proper encapsulation and no name collisions thanks to scoping, many confusing sources of problems are eliminated. However, Typst's strictness also forces users to do things the right way—even though a simpler, more frail way might actually work for their use case. Maybe they wanted to write a simple `cite` function that takes a citation key and pushes it into a global array that is read and formatted in the end. This does not work in Typst for good reason: Typst guarantees that extracting a piece of content into a variable lets that variable behave just like the content in every regard. This holds true no matter whether the variable is used zero times, once, or multiple times. This is a valuable guarantee because it ensures that content is, in fact, composable and that it can be transparently handled as a computational value without any unexpected consequences. Impure functions like the `cite` function above violate this guarantee as, depending on when and how often the function would be called, the length and order of the array would differ.

In principle, macros sidestep this issue because they simply expand to their definition and thus behave just like it. However, they come with their own set of related problems, mostly stemming from the interaction of macro expansion and execution. TEX makes a distinction between macros and primitives: When the next token is a macro, TEX expands it and when it is a primitive, TEX executes it. Primitives can depend on and modify state, for example through registers. This leads to problems when LATEX writes commands to a file to read them in the next compilation (e.g., for the table of contents). Then, TEX operates in expansion-only mode, breaking macros whose expansion depends on side effects of the execution of primitives within them. These macros are called *fragile*. Most LATEX macros have been made *robust* (that is, not fragile) over time but not all of them. For example, the code below does not produce the desired output in LATEX. Instead of having a stacked $\substack{a\\b}$ in both the table of contents and the section heading, the table of contents just breaks in weird ways. The fix for this is adding a `\protect` command before the fragile `\substack` command. Typst, on the other hand, *forces* users to do certain things the right way. This takes away from its simplicity, but it also solves confusing problems.

```latex
\tableofcontents                                                LATEX
\section{$\substack{a\\b}$} % Fix: Add \protect
```

Diagnostics. A direct feedback loop is central to learning anything new. For markup and programming languages, error messages are the primary way users get feedback—making them incredibly important. Good error messages are readable and actionable. [62] Table 4 illustrates three different examples of erroneous code alongside the diagnostics produced by the Typst and TEX compilers.

1. The first example demonstrates TEX's interactive error correction: Instead of complaining about a missing dollar sign, TEX tells the user that it inserted the missing dollar sign and asks whether it should proceed (with crossed fingers). [3] The reason for this design is that, when TEX was written, a single compilation could take minutes and restarting at every small mistake would have been a waste of time. Even so, on today's machines, compilations are fast and users do not expect their compiler to ask them how to proceed. Although TEX compilers have a command line flag (`-interaction=nonstopmode`) to disable the interactivity, the error messages stay the same and are thus still written from the error-correcting perspective.

2. In the second example, we can see the benefits of well-defined syntactical expressions and a proper type system. Typst can tell the user both what it wanted (a length) and what it got (content). In contrast, TEX could only tell the user that it expected a number and that it did not find a legal unit of measure. It neither tells the user what it found instead (which can help immensely with understanding an error) nor that it wanted a *dimension* or *skip*. Making this connection from the two individual error messages is up to the user.

Typst	TeX/LaTeX
`$x+y` `    ^` expected closing dollar sign	`$x+y` Missing `$` inserted.
`#set par(leading: [Hello])` `                  ^^^^^^^` expected length, found content	`\baselineskip=Hello` Missing number, treated as zero. Illegal unit of measure (pt inserted).
`#heading()` `        ^^` missing argument: body (Writing just `#heading` is not an error, it simply prints out `<function heading>`.)	`\section` Missing `\endcsname` inserted. Missing `\endcsname` inserted. ... (The above error repeats 100 times, then LaTeX gives up.)

Table 4: Comparison of Typst and TeX diagnostics. The TeX/LaTeX error messages were produced with local `pdftex/pdflatex` compilers and the command line flag `-interaction=nonstopmode`.

3. The third example illustrates the problems TeX faces when there are multiple layers of macro abstractions (in this case, macros from LaTeX). It compares how Typst and TeX react when a section heading's text is missing. Typst complains that an argument called "body" is missing. If the user is familiar with the terms *argument* and *body,* this error message is quite clear. (Typst consistently refers to the content contained within some element as its *body.*) TeX typically complains with "Runaway argument?" when an argument is missing. Not in this case though, instead LaTeX gets stuck and repeats the same error over and over again: "Missing `\endcsname` inserted." (When disabling the interactive mode, it stops after 100 errors.) This error message is simply not actionable.

Since there is not one standard XML processor we could test, XML was left out of this comparison. However, thanks to its well-defined syntax and support for validatable schemas, the preconditions for excellent error message are fulfilled. Similarly, HTML and CSS are left out because the primary applications consuming these documents (browsers) do not produce error messages for them.

8.2 Automatability

In typesetting, automatability arises in two ways: Through strong *computational foundations* and through *supplemental systems* that facilitate automation in the presence of specific patterns. The computational foundations let users do arbitrary computation, but, maybe even more importantly, they provide an interface for using systems in a coherent way. Systems achieve automation by exploiting patterns in the translation from input to output. For example, a structure-based styling system lets users automate the way certain structural elements are styled, making use of the fact that typically all top-level section headings in a document look the same. Similarly, Typst's introspection system lets users automatically create pieces of the document that depend on the remaining document's structure. Where there are patterns, there can be systems—and patterns are everywhere.

Foundations. The computational foundations of a markup language form the basis for everything built on top. Typst's computational layer provides data structures and imperative programming constructs. Most data structures are well-known from other programming languages (strings, arrays and dictionaries) while *content* is specific to typesetting. Manipulation and composition of these data structures primarily occurs through methods and joining, respectively.

TeX has no proper data structures: Everything is a list of tokens. Because even basic manipulation of token list data structures is so complex, almost every piece of programming in TeX makes use of some package. The first example of Table 5 shows how to trim a string with the package `trimspaces`. Similarly, the second example shows that splitting a string by commas and iterating over the items is a task for a package (`listofitems`). Knuth himself describes that he did not want TeX to become a fully fledged programming language. He only put in programming constructs little by little, partly due to urging from other people. [4] Nonetheless, in the end, people used and still use TeX as a programming language. The third example of Table 5 compares very basic programming constructs in Typst and TeX. Especially the `\ifnum` command that combines the if construct with an integer comparison illustrates the lack of principled constructs.

Despite all this, TeX is more flexible and customizable than Typst. Since macros are not governed by the normal rules of scoping, it is possible to redefine or hook into most things. However, breaking encapsulation in this way also comes with the possibility of conflict (when multiple packages override an internal macro of another package). Furthermore, it limits the ability of package developers to update and improve the internals of their packages. Typst functions behave in the opposite way. They are encapsulated and therefore less customizable. But in return, they define proper interfaces for controlled customization, sidestepping package conflicts. To summarize programming in TeX: *Anything is possible and everything is complicated.*

LuaTeX aims to improve this dire situation. It lets users embed arbitrary Lua code into their TeX files and provides interfaces to TeX's core. As Lua is a well-known, capable scripting language, this is in principle a great solution. Through LuaTeX's capable API, users have utmost freedom in customizing and modifying TeX (far more than they have with Typst). The domain where LuaTeX falls short in comparison to Typst is content construction and composition. While Typst has an encapsulated content data type, LuaTeX only has strings and token lists. Inserting content into the document involves the `tex.print` function. [63]

HTML and XML themselves do not provide programming facilities. However, HTML has JavaScript and through the *DOM* (*Document Object Model*) it can create and manipulate HTML nodes. In-memory DOM nodes that are not hooked into the document are similar to Typst's content values. However, working with DOM nodes requires far more boilerplate than with Typst content since JavaScript does not have special syntax and semantics specifically for DOM manipulation. Besides, DOM nodes cannot integrate local CSS rules like Typst content can with set and show rules. Meanwhile, XML has XSL, which has a programming model built on pure transformations. By virtue of being an XML dialect, it is extremely verbose and not well-suited for non-trivial programs.

Typst	**TeX**

`#{"Hello  ".trim(at: end)}`	`\usepackage{trimspaces}`
	`\trim@post@space{Hello  }`

`#let tabelize(str) = {`
`  let animals = str.split(", ")`
`  table([*Animal*], ..animals)`
`}`
 	
`#tabelize(`
`  "Tiger, Giraffe, Cougar"`
`)`
	`\usepackage{listofitems}`
	`\def\tabelize#1{`
	`  \readlist\animals{#1}`
	`  \begin{table}`
	`    \textbf{Animal} \\`
	`    \foreachitem\a\in\animals{`
	`      \a \\`
	`    }`
	`  \end{table}`
	`}`
	
	`\tabelize{Tiger, Giraffe, Cougar}`

`#let i = 0`
`#while i < 5 {`
`  i += 1`
`  [Hello #i. ]`
`}`
	`\newcount\i \i=0`
	`\loop`
	`  \advance \i by 1`
	`  Hello \the\i.`
	`\ifnum \i<5 \repeat`

Table 5: Comparison of programming capabilities in Typst and TeX/LaTeX.

Supplemental Systems. Powerful systems like CSS don't necessarily need to be built on computational foundations. However, strong foundations *supercharge* them. A good example for this is the following: In CSS, the `list-style-type` [25] property defines how to label a list. It has numerous options for circles, squares, roman, arabic, and chinese numbers. However, this was still not enough to fulfill all users' needs, so CSS gained support for `@counter-style` rules [64] that allow the definition of custom labelling styles (albeit still in a limited form). Typst, on the other hand, simply allows users to set the list labelling style to a function mapping from an integer to arbitrary content. By integrating the styling system with the computational layer, it becomes simpler and more flexible.

On one side of the spectrum, there are general programming languages: These have strong computational foundations, but they lack the systems for typesetting. On the other side we have domain-specific languages like CSS: They have the necessary systems to perform certain automations but cannot make the best use of them because they lack the computational foundations. Typst is a mix of both: It integrates a computational layer built for typesetting with systems for structure and styling. Not only does this make the systems simpler and more expressive, it also presents an opportunity for unification: JavaScript and XSL each have their own, different ways to check conditions whereas Typst has a single `if` expression. With this in mind, we will now compare Typst's systems for structuring, introspection, and styling to those of the alternatives.

Structuring. A system that can gracefully encode a document's structure is very valuable. It enables structure-based styling and introspection, and makes it possible to produce documents that are accessible to people with visual impairment. XML is the strongest contender in this domain as, given the right schema, the author has no option but to write truly descriptive markup. Furthermore, there is a rich ecosystem of XML processors and standardized XML formats for books, articles, and more. While LaTeX is designed around the idea of structure, the structure is only present in the front-end. Before layout, all structural information is lost, making the resulting PDFs inaccessible. Not all hope is lost though: In 2020, the LaTeX project announced the “Tagged PDF project” with the aim to improve this situation. [42] Typst’s structural system does not yet integrate with user-defined functions and our current compiler, like LaTeX, loses the structural information during typesetting. However, these limitations are easier to overcome for Typst than LaTeX because Typst is not limited by the constraints of TeX and its macro system.

Introspection. Typst’s introspection system is quite unique: While LaTeX also moves information around to build things like the table of contents, Typst’s introspection is (a) fully automatic and (b) integrated with the structural system. In LaTeX, writing an introspection for a certain kind of structural element involves changing the definition of that element so that it writes information to a file. The introspection can then read this information from the file in the next compilation. Users need to recompile manually and they might not even know how many compilations are needed for all results to stabilize. In contrast, Typst automatically relayouts until all introspections stabilize. Furthermore, the definition of a structural element need not even be changed for it to be inspectable in Typst—all structural elements are inspectable by default. This leads to simpler introspections and less coupling between elements and introspections. Although XSL can also generate a table of contents from an XML file, this is a different scenario as it gives no access to layout-stage information like page numbers.

Styling. Last but not least is the styling system, the most important individual system for typesetting. In CSS, multiple documents can share one style sheet and one document can be styled by multiple style sheets. In contrast, multiple independent XSL style sheets are not easily composable. An XSL style sheet transforms XML following a very specific schema. While it is possible to apply one XSL transformation after another, most likely the output of the first one will not have the structure the second one expects. In Typst, both property-based and transformational styles from different sources compose well. When different “style sheets” set a value for the same property or define a show rule recipe for the same element, the more local style wins. With recursive show rules, it is even possible that both can come into play.

Typst scopes properties by element. This way, it is not only clear which properties apply to an element, the properties also become more discoverable: The user can expressly look up all properties of a specific element when styling that element. In CSS, on the contrary, all properties live in a shared namespace and without consulting the documentation a user cannot know which properties apply to which elements. Even though XSL also supports property-based styling, it is much more complicated than CSS, such that even the W3C says, “Use CSS when you can, use XSL when you must.” [65] In contrast to Typst, both CSS and XSL have powerful contextual selection mechanisms. Typst can currently only select elements by type, while CSS can select by ancestry, siblingship, and attribute.

Table 6 shows how to style an ordered list locally or globally with Typst, TeX, and CSS. More local styling is useful for quickly putting something together and for local

	Typst	CSS	TeX
Local	`#enum(` `label: "I.",` `[Rome],` `[Alexandria],` `)`	`ol.named {` `list-style-type:` `upper-roman;` `}` (Rome and Alexandria are part of the HTML.)	`\begin{enumerate}` `[label=\Roman*.]` `\item Rome` `\item Alexandria` `\end{enumerate}`
Global	`#set enum(label: "I.")`	`ol {` `list-style-type:` `upper-roman;` `}`	`\setenumerate[0]` `{label=\Roman*.}`

Table 6: Local and global styling in Typst, LaTeX and CSS

one-time fixes. Global styling leads to cleaner, more descriptive markup and increases overall flexibility. By scoping set and show rules, in Typst, styles can be as local or global as necessary. In HTML, elements have a class attribute through which CSS rules can select specific elements to perform local styling. The downside of this scheme is that the user needs to come up with a unique class name for every local style. TeX has no proper styling system. Documents are styled by calling or redefining certain commands. Because it has no styling system, TeX also does not have a general mechanism for styling something locally or globally. Most of the time, there are simply two macros for the two cases. For example, there is `\textbf{..}` and `\bfseries ...` The former renders a piece of text in bold face while the latter makes the primary font bold for the remainder of the group/document. Similarly, the popular `enumitem` package provides a local and a global way to configure the enumeration label.

Chapter 9

Conclusion

Markup languages have tangible benefits compared to Wysiwyg typesetting solutions. They let authors express their document's structure independently from its appearance and provide the tools to automate the conversion from one to the other. This leads to a better writing experience and less manual work. However, existing markup languages for typesetting are unsatisfactory: While TEX-based solutions produce very high-quality output, they are difficult to use, produce poor error messages, and are plagued by package conflicts. XML-based approaches work well for highly scaled publisher operations but are unsuitable for direct authoring and document-local automation.

We have presented a new programmable markup language for typesetting called *Typst* and discussed its design from a user-facing and an internal perspective. Typst's goal is to be simple, yet highly automatable. Typst achieves this goal through consistent language design. On the syntactical side, Typst is familiar both for users of lightweight markup languages and for programmers. Building its evaluation model around pure functions instead of macros leads to more predictable results, less conflicts, and actionable error messages. It also allows for very performant implementations of Typst. Meanwhile, the ability to handle content as a programmatic value facilitates abstraction and encapsulation, and thereby automation. By grouping style properties around elements instead of having a CSS-like shared namespace, properties become more discoverable and their domains clearer. The fact that style properties may also be functions (e.g., the `label` property of ordered lists) gives rise to evermore flexibility.

We have also implemented a compiler for Typst that already handles most of the language as discussed in this thesis. Both the Typst language and its compiler are still evolving. Typst's primary limitation compared to existing tools is that its styling system does not yet support selection based on ancestry and neighborhood relations. Finding the best way to integrate such contextual styling with Typst's existing systems is a challenging opportunity. Further goals include a generalization of the layout model and more accessible PDF output.

Typst is not purely a research project: While this thesis discusses Typst from an academic point of view, we truly want to simplify markup-based typesetting for users from around the world. Part of this effort is the creation of a web-based editing environment with helpful tooling and instant preview.[3] Typst's journey has only just started.

[3] The project's home page is https://typst.app.

References

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible Markup Language (XML) 1.0,” W3C, 2008. Accessed: Aug. 29, 2022. Online. Available: https://www.w3.org/TR/xml/

[2] J. Gruber, “Markdown,” *Daring Fireball.* https://daringfireball.net/projects/markdown/ (accessed Aug. 16, 2022).

[3] D. E. Knuth, *The TeXbook.* Reading, MA, USA: Addison-Wesley, 1986.

[4] “Amsterdam, 13 March 1966 — Knuth meets NTG members,” *TUGboat,* vol. 17, no. 4, pp. 342–355, Dec. 1996.

[5] L. Lamport, *LaTeX: A document preparation system,* 2nd ed. Reading, MA, USA: Addison-Wesley, 1994.

[6] CTAN, “CTAN: Comprehensive TeX archive network.” https://ctan.org/ (accessed Aug. 29, 2022).

[7] F. Mittelbach, “E-TeX: Guidelines for future TeX extensions,” *TUGboat,* vol. 11, no. 3, pp. 86–94, May 1993.

[8] M. Haug, “Fast typesetting with incremental compilation,” M.S. thesis, Tech. Univ. Berlin, Berlin, Germany, 2022.

[9] B. W. Kernighan, *UNIX: A history and a memoir.* Kindle Direct Publishing, 2020.

[10] J. F. Ossanna and B. W. Kernighan, “Troff user’s manual,” AT&T Bell Laboratories, Computer Science Technical Report No. 54, 1992.

[11] B. W. Kernighan, “A TROFF tutorial,” Aug. 1978.

[12] CTAN, “What are TeX and its friends?,” *CTAN: Comprehensive TeX Archive Network.* https://www.ctan.org/tex/ (accessed Aug. 09, 2022).

[13] D. E. Knuth and M. F. Plass, “Breaking paragraphs into lines,” *Softw: Pract. Exper.,* vol. 11, no. 11, pp. 1119–1184, Nov. 1981, doi: 10.1002/spe.4380111102.

[14] P. Flynn, “Rolling your own document class: Using LaTeX to keep away from the dark side,” *TUGBoat,* vol. 28, no. 1, pp. 110–123, 2007.

[15] C. F. Goldfarb, “The roots of SGML – A personal recollection,” 1996. Accessed: Aug. 02, 2022. Online. Available: http://www.sgmlsource.com/history/roots.htm

[16] C. F. Goldfarb, “A generalized approach to document markup,” *SIGPLAN Not.,* vol. 16, no. 6, pp. 68–73, Apr. 1981, doi: 10.1145/872730.806456.

[17] M. Goossens and J. Saarela, “A practical introduction to SGML,” presented at the TUG95, Saint-Petersburg, FL, USA, Jul. 1995.

[18] “Information processing — Text and office systems — Standard Generalized Markup Language (SGML),” ISO Standard 8879:1986, 1986.

[19] H. W. Lie and J. Saarela, “Multipurpose web publishing using HTML, XML, and CSS,” *Commun. ACM,* vol. 42, no. 10, pp. 95–101, Oct. 1999, doi: 10.1145/317665.317681.

[20] “A history of HTML,” in *Raggett on HTML 4,* 2nd ed., Reading, MA, USA: Addison-Wesley, 1998. Accessed: Aug. 02, 2022. Online. Available: https://www.w3.org/People/Raggett/book4/ch02.html

[21] J. Clark, “Comparison of SGML and XML,” World Wide Web Consortium, Dec. 1997. Accessed: Aug. 09, 2022. Online. Available: https://www.w3.org/TR/NOTE-sgml-xml-971215/

[22] M. Hilbert, A. Witt, and O. Schonefeld, “Making CONCUR work,” in *Proc. Extreme Markup Lang.,* Montréal, Canada, Aug. 2005, pp. 1–18.

[23] H. W. Lie, “Cascading HTML style sheets – A proposal,” Oct. 1994. Accessed: Aug. 16, 2022. Online. Available: https://www.wiumlie.no/2006/phd/archive/www.w3.org/People/howcome/p/cascade.html

[24] H. W. Lie and B. Bos, “Cascading Style Sheets, level 1,” W3C, 1996. Accessed: Aug. 29, 2022. Online. Available: https://www.w3.org/TR/REC-CSS1/

[25] B. Bos, “Cascading Style Sheets level 2 revision 2 (CSS 2.2) specification,” W3C, 2016. Accessed: Aug. 30, 2022. Online. Available: https://www.w3.org/TR/CSS22/

[26] H. W. Lie, “Cascading Style Sheets,” Ph.D. dissertation, University of Oslo, Oslo, Norway, 2005.

[27] J. Bosak, “DSSSL online application profile,” 1996. Accessed: Aug. 29, 2022. Online. Available: https://www.ibiblio.org/pub/sun-info/standards/dsssl/dssslo/do960816.htm

[28] A. Berglund *et al.,* “XML Path Language (XPath) 2.0,” W3C, 2010. Accessed: Aug. 17, 2022. Online. Available: https://www.w3.org/TR/xpath20/

[29] J. van Ossenbruggen, L. Hardman, L. Rutledge, and A. Eliëns, “Style sheet languages for hypertext,” *SIGWEB Newslett.,* vol. 6, no. 3, pp. 16–20, Oct. 1997, doi: 10.1145/288190.288193.

[30] S. Leonard, “The text/markdown media type,” IETF, RFC 7763, Mar. 2016. doi: 10.17487/RFC7763.

[31] J. Gruber, “Markdown: Syntax.” Accessed: Aug. 06, 2022. Online. Available: https://daringfireball.net/projects/markdown/syntax

[32] Eclipse Foundation, “AsciiDoc.” https://asciidoc.org/ (accessed Sep. 01, 2022).

[33] P. Jipsen and D. Lippman, “AsciiMath.” http://asciimath.org/ (accessed Aug. 16, 2022).

[34] J. MacFarlane, “Pandoc: A universal document converter.” https://pandoc.org/ (accessed Aug. 29, 2022).

[35] J. Kew, “XƎTEX,” CTAN: *Comprehensive TEX Archive Network.* https://tug.org/xetex/ (accessed Sep. 01, 2022).

[36] T. Hoekwater, H. Henkel, and H. Hagen, “LuaTEX.” https://www.luatex.org/ (accessed Sep. 01, 2022).

[37] P. Gundlach, “Speedata.” https://www.speedata.de (accessed Sep. 01, 2022).

[38] M. Flatt and E. Barzilay, “Scribble: The Racket Documentation Tool,” *GitHub.* https://github.com/racket/scribble (accessed Sep. 01, 2022).

[39] RStudio, “Quarto.” https://quarto.org/ (accessed Sep. 01, 2022).

[40] S. Cozens and C. Maclennan, “The SILE typesetter.” https://sile-typesetter.org/ (accessed Aug. 29, 2022).

[41] F. Hatat *et al.,* “Patoline: A modern digital typesetting system.” https://patoline.github.io/ (accessed Aug. 16, 2022).

[42] F. Mittelbach, “Quo Vadis LATEX(3) — A look at the upcoming years,” presented at the TUG 2020, Sep. 12, 2020. Accessed: Aug. 12, 2022. Online. Available: https://www.youtube.com/watch?v=zNci4lcb8Vo

[43] F. Mittelbach, “LATEX’s hook management,” Jul. 2022. Accessed: Aug. 29, 2022. Online. Available: https://mirrors.ctan.org/macros/latex/base/lthooks-code.pdf

[44] J. H. Coombs, A. H. Renear, and S. J. DeRose, “Markup systems and the future of scholarly text processing,” *Commun. ACM,* vol. 30, no. 11, pp. 933–947, Nov. 1987, doi: 10.1145/32206.32209.

[45] M. D. Ernst, G. J. Badros, and D. Notkin, “An empirical analysis of C preprocessor use,” *IEEE Trans. Softw. Eng.,* vol. 28, no. 12, pp. 1146–1170, Dec. 2002, doi: 10.1109/TSE.2002.1158288.

[46] National Library of Medicine, “Journal Article Tag Suite.” https://jats.nlm.nih.gov/ (accessed Aug. 09, 2022).

[47] National Library of Medicine, “Book Interchange Tag Set.” https://jats.nlm.nih.gov/extensions/bits/ (accessed Aug. 09, 2022).

[48] M. Kohm, *KOMA-Script: Ein wandelbares LaTeX2ε-Paket,* 7th ed. Berlin, Germany: DANTE e.V., Lehmanns Media, 2020.

[49] J. Johnson and R. J. Beach, “Styles in document editing systems,” *Computer,* vol. 21, no. 1, pp. 32–43, Jan. 1988, doi: 10.1109/2.222115.

[50] E. J. Etemad and T. Atkins-Bittner, “Selectors level 4,” W3C, 2022. Accessed: Aug. 29, 2022. Online. Available: https://drafts.csswg.org/selectors/#relational

[51] M. Kay, “XSL Transformations (XSLT) Version 2.0,” W3C, 2021. Accessed: Aug. 10, 2022. Online. Available: https://www.w3.org/TR/2021/REC-xslt20-20210330/

[52] N. L. C. Talbot, “Auxiliary files,” in *LaTeX for complete novices,* Norfolk, UK: Dickimaw Books, 2012.

[53] WhatWG, “HTML living standard,” WHATWG. Accessed: Aug. 29, 2022. Online. Available: https://html.spec.whatwg.org/

[54] “IEEE standard for floating-point arithmetic,” IEEE Standard 754-2019, 2019.

[55] S. Klabnik and C. Nichols, *The Rust programming language.* San Francisco, CA, USA: No Starch Press, 2018.

[56] Apple Inc., “Array | Apple developer documentation.” Accessed: Aug. 29, 2022. Online. Available: https://developer.apple.com/documentation/swift/array

[57] M. Davis, A. Lanin, and A. Glass, “Unicode bidirectional algorithm,” Unicode, Inc., UAX #9, 2021. Accessed: Aug. 29, 2022. Online. Available: https://unicode.org/reports/tr9/

[58] T. Atkins-Bittner, E. J. Etemad, and R. Atanassov, “CSS grid layout module level 2,” W3C, 2020. Accessed: Aug. 29, 2022. Online. Available: https://www.w3.org/TR/css-grid-2/

[59] T. Newsham, “Introduction to Haskell IO/Actions,” *HaskellWiki.* Accessed: Aug. 29, 2022. Online. Available: https://wiki.haskell.org/Introduction_to_Haskell_IO/Actions

[60] D. D. McCracken and E. D. Reilly, “Backus-Naur Form (BNF),” in *Encyclopedia of Computer Science,* Hoboken, NJ, USA: John Wiley and Sons, 2003, pp. 129–131.

[61] M. Hills, P. Klint, T. van der Storm, and J. Vinju, “A case of visitor versus interpreter pattern,” in *Lecture Notes Comput. Sci.,* Berlin, Heidelberg, Germany, 2011, vol. 6705, pp. 228–243. doi: 10.1007/978-3-642-21952-8_17.

[62] P. Denny *et al.,* “On designing programming error messages for novices: Readability and its constituent factors,” in *Proc. 2021 CHI Conf. Human Factors Comput. Syst.,* New York, NY, USA, May 2021, pp. 1–15. doi: 10.1145/3411764.3445696.

[63] LuaTeX development team, “LuaTeX reference manual,” 2022. Accessed: Sep. 01, 2022. Online. Available: http://mirrors.ibiblio.org/CTAN/systems/doc/luatex/luatex.pdf

[64] T. Atkins-Bittner, “CSS counter styles level 3,” W3C, 2021. Accessed: Aug. 22, 2022. Online. Available: https://www.w3.org/TR/css-counter-styles-3/

[65] B. Bos, “CSS and XSL: Which should I use?,” W3C, Jul. 1999. Accessed: Aug. 25, 2022. Online. Available: https://www.w3.org/Style/CSS-vs-XSL.en.html

Appendix A: Typst Grammar

Below is an approximate EBNF grammar for the Typst language that is based on our hand-written recursive descent parser. We follow these conventions:

- Production names are all lowercase.
- Text enclosed in single (`'`) or double quotes (`"`) defines a terminal.
- `*` for an arbitrary number of repetitions.
- `+` for at least one repetition.
- `?` for zero or one repetitions.
- `!` to negate a simple (character-class-like) production.
- `.` to match an arbitrary character.
- `a - b` to match anything that matches `a` but not `b`.
- `unicode(Property)` to match any character that has the given unicode property.

Note that comments and spaces are allowed almost everywhere within code constructs. For readability, this is omitted in the grammar. Moreover, the grammar omits the indentation rules for lists as EBNF cannot handle context-sensitive constructs. [60]

```
// Markup.
markup ::= markup-node*
markup-node ::=
  space | linebreak | text | escape | nbsp | shy | endash | emdash |
  ellipsis | quote | strong | emph | raw | link | math | heading |
  list | enum | desc | label | ref | markup-expr | comment

// Markup nodes.
space ::= unicode(White_Space)+
linebreak ::= '\' '+'?
text ::= (!special)+
escape ::= '\' special
nbsp ::= '~'
shy ::= '-?'
endash ::= '--'
emdash = '---'
ellipsis ::= '...'
quote ::= "'" | '"'
strong ::= '*' markup '*'
emph ::= '_' markup '_'
raw ::= '`' (raw | .*) '`'
link ::= 'http' 's'? '://' (!space)*
math ::= ('$' .* '$') | ('$[' .* ']$')
heading ::= '='+ space markup
list ::= '-' space markup
enum ::= digit* '.' space markup
desc ::= '/' space markup ':' space markup
label ::= '<' ident '>'
ref ::= '@' ident
markup-expr ::= block | ('#' hash-expr)
hash-expr ::= ident | func-call | keyword-expr
```

```
special ::=
  '\' | '/' | '[' | ']' | '{' | '}' | '#' | '~' | '-' | '.' | ':' |
  '"' | "'" | '*' | '_' | '`' | '$' | '=' | '<' | '>' | '@'

// Code and expressions.
code ::= (expr (separator expr)* separator?)?
separator ::= ';' | unicode(Newline)
expr ::=
  literal | ident | block | group-expr | array-expr | dict-expr |
  unary-expr | binary-expr | field-access | func-call | method-call |
  func-expr | keyword-expr
keyword-expr ::=
  let-expr | set-expr | show-expr | wrap-expr | if-expr |
  while-expr | for-expr | import-expr | include-expr |
  break-expr | continue-expr | return-expr

// Literals.
literal ::= 'none' | 'auto' | boolean | int | float | numeric | str
boolean ::= 'false' | 'true'
int ::= digit+
float ::= ((digit+ ('.' digit*)?) | ('.' digit+)) ('e' digit+)?
numeric ::= float unit
digit = '0' | ... | '9'
unit = 'pt' | 'mm' | 'cm' | 'in' | 'deg' | 'rad' | 'em' | 'fr' | '%'
str ::= '"' .* '"'

// Identifiers.
ident ::= (ident_start ident_continue*) - keyword
ident_start ::= unicode(XID_Start) | '_'
ident_continue ::= unicode(XID_Continue) | '_' | '-'
keyword ::=
  'none' | 'auto' | 'true' | 'false' | 'not' | 'and' | 'or' |
  'let' | 'set' | 'show' | 'wrap' | 'if' | 'else' | 'for' | 'in' |
  'as' | 'while' | 'break' | 'continue' | 'return' | 'import' |
  'include' | 'from'

// Blocks.
block ::= code-block | content-block
code-block ::= '{' code '}'
content-block ::= '[' markup ']'

// Groups and collections.
group-expr ::= '(' expr ')'
array-expr ::= '(' ((expr ',') | (expr (',' expr)+ ','?))? ')'
dict-expr ::= '(' (':' | (pair (',' pair)* ','?)) ')'
pair ::= (ident | str) ':' expr
```

```
// Unary and binary expression.
unary-expr ::= unary-op expr
unary-op ::= '+' | '-' | 'not'
binary-expr ::= expr binary-op expr
binary-op ::=
  '+' | '-' | '*' | '/' | 'and' | 'or' | '==' | '!=' |
  '<' | '<=' | '>' | '>=' | '=' | 'in' | ('not' 'in') |
  '+=' | '-=' | '*=' | '/='

// Fields, functions, methods.
field-access ::= expr '.' ident
func-call ::= expr args
method-call ::= expr '.' ident args
args ::= ('(' (arg (',' arg)* ','?)? ')' content-block*) | content-block+
arg ::= (ident ':')? expr
func-expr ::= (params | ident) '=>' expr
params ::= '(' (param (',' param)* ','?)? ')'
param ::= ident (':' expr)?

// Keyword expressions.
let-expr ::= 'let' ident params? '=' expr
set-expr ::= 'set' expr args
show-expr ::= 'show' (ident ':')? expr 'as' expr
wrap-expr ::= 'wrap' ident 'in' expr
if-expr ::= 'if' expr block ('else' 'if' expr block)* ('else' block)?
while-expr ::= 'while' expr block
for-expr ::= 'for' for-pattern 'in' expr block
for-pattern ::= ident | (ident ',' ident)
import-expr ::= 'import' import-items 'from' expr
import-items ::= '*' | (ident (',' ident)* ','?)
include-expr ::= 'include' expr
break-expr ::= 'break'
continue-expr ::= 'continue'
return-expr ::= 'return' expr?

// Comments.
comment = line-comment | block-comment
line-comment = '//' (!unicode(Newline))*
block-comment = '/*' (. | block-comment)* '*/'
```