Technische Universität Berlin
Faculty IV – Electrical Engineering and Computer Science

Master's thesis

Fast Typesetting with Incremental Compilation

Martin E. Haug

June 2022
First Examiner: Sohan Lal
Second Examiner: Sabine Glesner
Thesis Advisor: Nicolai Stawinoga

Martin Haug
Matr.-Nr. [redacted]
Major: M.Sc. Computer Science
m.haug@tu-berlin.de

Affidavit on the Originality of this Thesis

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:

Text-based typesetting systems like LaTeX compile source files into PDF and other formats. This process is computationally expensive. Unlike "What You See Is What You Get" (WYSIWYG) systems, most text-based typesetting systems recompile the document from scratch, even after minor changes. Constant expensive recomputations prevent instant preview of changes, as in WYSIWYG software. This thesis explores how to speed up the text-based typesetting software *Typst* by reusing artifacts from past compilations while parsing and layouting. We propose an incremental parsing algorithm that accommodates context-sensitive properties of markup languages such as the "off-side rule" and is suited for handwritten recursive descent parsers. Furthermore, we also present a constrained layout caching scheme that allows reusing previous layout elements even if the dimensions of their parent containers changed. In an evaluation based on a set of typical documents and edits, we determined that this thesis' contributions increase the speed of Typst by a factor between 4.5 and 91. Compared with LaTeX, Typst can compile an edit between 3.4 and 9895 times as fast.

Zusammenfassung:

Textbasierte Satzsysteme wie LaTeX kompilieren Quelldateien in PDF-Dateien und andere Formate. Dieser Prozess ist rechenaufwändig. Im Gegensatz zu „What You See Is What You Get"-Systemen (WYSIWYG) kompilieren die meisten textbasierten Satzsysteme das Dokument von Grund auf neu, sogar nach kleinen Änderungen. Die ständigen langsamen Neuberechnungen verhindern eine sofortige Vorschau nach Änderungen, wie sie WYSIWYG-Software bietet. In dieser Masterarbeit wird untersucht, wie die textbasierte Satzsoftware *Typst* durch die Wiederverwendung von Artefakten aus früheren Kompilierungen beim Parsen und Layouten beschleunigt werden kann. Wir präsentieren einen inkrementellen Parsing-Algorithmus, der kontextsensitive Eigenschaften von Markup-Sprachen wie die „off-side rule" berücksichtigt und für manuell implementierte Recursive Descent-Parser geeignet ist. Darüber hinaus stellen wir ein Layout-Caching-Schema vor, das die Wiederverwendung früherer Layout-Elemente ermöglicht, selbst wenn sich die Abmessungen ihrer übergeordneten Container geändert haben. In einer Evaluierung, die auf einer Reihe von typischen Dokumenten und Bearbeitungen basiert, haben wir festgestellt, dass die Beiträge dieser Arbeit die Geschwindigkeit von Typst um einen Faktor zwischen 4,5 und 91 erhöhen. Im Vergleich zu LaTeX kann Typst eine Änderung zwischen 3,4 und 9895 mal so schnell verarbeiten.

Table of Contents

1 Introduction

Computer typesetting is a domain with many intriguing problems for algorithm designers and software architects, but it has not seen much innovation in the last decade. The text-based typesetting software LaTeX has enjoyed a strong position in the sciences for the last 30 years. This system composes attractive page layouts but lacks speed, usability, and support for modern text encodings and non-Latin scripts.

The text-based typesetting system *Typst,* in whose development the author is involved, aims to address these issues. Typst is a newly developed system that combines the text-based workflow known from TeX with the interactive user experience typical for "What You See is What You Get" Desktop Publishing Software. Users of Typst shall receive an instant visual preview of the result of each keystroke. Hence, the Typst compiler must be fast to process new changes, departing from the offline and batch mode paradigm of TeX. However, since text-based typesetting software must perform expensive calculations when laying out a document, this goal cannot be reached without reusing artifacts from previous compilations.

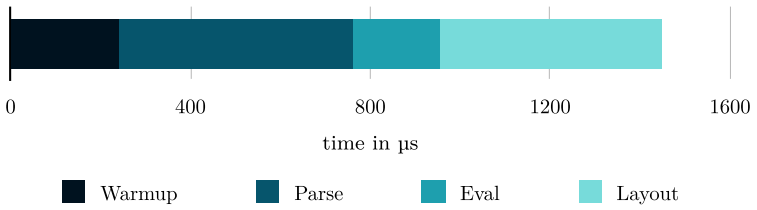


Figure 1: Wall clock time each stage of the Typst compiler needs for an 20-page document (cf. "Brochure" in Section 6.1).

We decided to tackle this problem from two angles: parsing and the main layout algorithm. Figure 1 shows how much time each stage of the Typst compiler needs when processing a 20-page document non-incrementally. Measurements for the backends that export to multiple targets (PDF, raster images, …) are not included in the figure as their various individual optimisations are not within the scope of this thesis. As can be seen from the figure, the layout stage is the most expensive stage of the Typst compiler and thus an obvious candidate for optimization. Parsing is another vital stage because it not only takes up a significant amount of time but also because it powers syntax highlighting, where speed is even more critical than for preview. Furthermore, the Abstract Syntax Tree becomes persistent and stable with incremental parsing, and changed nodes between edits can easily be identified, laying the groundwork for further optimizations. Hence, this thesis contributions focus on two areas: We propose incremental parsing and layouting schemes and evaluate both.

- *Parsing:* We present requirements for incremental parsing and a new incremental parsing algorithm that satisfies them. The algorithm is based on red-green trees that can easily be added to existing recursive descent parsers, requiring only lightweight modifications. It supports common *context-sensitive* properties of programming languages through its self-validation. Examination of the grammatical properties of markup languages and the engineering needs of parser implementations in various programming languages motivate research into such an algorithm.

- *Layouting:* We developed a comprehensive caching scheme that aims to retain and reuse results at every layer of a layout tree is proposed: Each element is layouted into some spacial region on a page. More than one similar region will yield the same layout result. Hence, each layout result is bundled with constraints, describing permissible regions. A cache then stores layout results and their constraints, and upon recompilation, a lookup mechanism returns results if the new regions satisfy their constraints. This scheme also includes a user-experience-focused approach for cache eviction where cache performance is optimized for frequent user operations like undo. The thesis also includes an empirical framework to establish the validity of constraints. We validate both algorithms by measuring their performance with a matrix of documents and edits.
- *Evaluation:* We implement these algorithms in Typst and conduct principled evaluation of both contributions individually and as a complete system, verifying whether they deliver the necessary speed improvements for an interactive document editing and preview experience. The Typst compiler with and without incremental compilation is also compared to several TEX compilers to determine the comparative speed differential.

The first important incremental parser paper, Ghezzi and Mandrioli 1979, is 43 years old. The field was actively developed until 2005, after which research interest waned. Recently, a resurgence in activity in this domain was spurred by impactful papers (e.g., Yedidia and Chong 2021). In the industry, the need for incremental parsers became more pronounced because of the development of better developer tooling like Syntax Tree-backed syntax highlighting, refactoring, and live code analysis. This push resulted in projects like Microsoft's .NET Compiler Platform (also known as *Roslyn*; tyoverby 2017) with an incremental parser as well as efforts in the *Rust* compiler. The latter compiler leverages a memoized query-based system during compilation (Lev et al. 2022), starting at the Abstract Syntax Tree. The validity of the memoized results is verified with a compilation-wide dependency graph (Woerister 2016).

Most of the research in this field has focussed on well-understood language classes like LR(1), LL(1), and even Generalized LR, which contains all context-free languages (Wagner 1998, Rosenkrantz and Stearns 1970, Brunsfeld 2021). Grammars in these classes are typically parsed bottom-up, making incremental computation easier since the prior state of the parser can be more easily recovered from the parse tree (see Section 3.1). However, these results cannot be directly applied to Typst's parser because of two reasons:

1. Typst, like many recent markup languages (cf. Gruber 2004)) is designed to leverage indentation to make the source file easier to read and write, especially when using structures like nested lists. However, languages with indentation sensitive blocks (known as the off-side rule; Landin 1966) cannot be parsed by a context-free parser like the ones above (Erdweg et al. 2013).
2. Additionally, all the above research has gone into parser generators, primarily for bottom-up parsing. However, parsing generators in general and bottom-up parsing specifically produce sub-par error messages. Since Typst aims to be usable by non-programmers and programmers alike, it uses a handwritten recursive descent parser.

The incremental parser in this thesis is uniquely suited for this problem: It can accommodate the off-side rule and other context-sensitive grammatical constructs and

complements handwritten recursive descent parsers without coupling too tightly with their code. Implementors only need to make two adjustments to adapt the algorithm to their language. This approach retains the recursive descent parser's capabilities and well-formed error messages.

Reusing layout results is a challenge unique to text-based typesetting. Other software that renders graphics usually has to deal with one area for its layout, which may or may not be infinite on either axis. Typesetters, however, have to arrange their layouts in multiple finitely sized regions, the pages. Splitting up elements such that they span two pages and picking where to break between pages is a problem that other software does not need to solve.

Recently, web browser vendors have been driving research and development in incremental layouting. Browsers need to remain interactive even when scripts change a webpage's elements. To guarantee fast relayouts, they use a solution where they evaluate the style and content changes a script has caused for a loaded page and then trigger a set of "reflows" and "redraws" for various changes of nodes in the Document Object Model (Lewis 2021). This technology is not directly applicable to Typst because of the discontinuous nature of pages and because the Typst compiler is not responsible for drawing or printing its output.

A simple approach would be to memoize parts of the layout (e.g., paragraphs), not dissimilar to how other compilers use compilation units to limit the scope of a recompilation. When a compiler, like *clang,* uses compilation units, they compile files separately and then link them together. Files only get recompiled if they or the interfaces of another code unit they use change (The Clang Team 2022). In Typst's context, this would mean a relayout if the layout element's content or target region dimensions changed. However, this is insufficient because the region dimensions for many elements in a document regularly change. However, most of these changes do not result in a different layout, as, for a paragraph, the remaining height on the page makes no difference as long as the page remains sufficiently tall to fit it fully.

Instead, the core of our caching scheme is spacial *constraints* that allow the reuse of cached layout for its original region and many other regions that would result in the same layout. This optimizes the cache hit ratio and, thus, effectiveness for our use case.

The next chapter provides the reader with a background in computer typesetting systems, particular tasks that a typesetter must perform, modern text representations for computers, and the Typst language and compiler. The third chapter discusses related work on both topics to survey both fields' prior art and evolution. The fourth and fifth chapters will then detail the main contributions of the thesis, first incremental parsing, then the layout cache. The chapter on the layout cache also presents how to derive constraints of various common layout types. The sixth chapter, Evaluation, contains two different benchmarking procedures: It examines both contributions separately as well as together and contrasts them with LaTeX. The seventh chapter follows, containing ideas for further research into and around the two incremental algorithms presented in this thesis. Finally, a summary of the research and results concludes the thesis.

This thesis was written in Typst.

2 Background

Today, virtually all printed documents are prepared on a computer instead of manually setting them with movable type or typing them on a typewriter. For this reason, a wide range of Desktop Publishing products has become available. On the one hand, software like *Adobe InDesign* or *QuarkXPress* offers professional users a "What You See Is What You Get" (WYSIWYG) experience in which they can take fine-grained control of typesetting properties like ligatures, kerning, proportional numerals, and more. On the other hand, text processing applications like *Microsoft Word* and *Apple Pages* have started incorporating features previously exclusive to desktop publishing. Some of the tools needed to create more appealing text compositions in a modern text processor have thus become widely available. Still, these applications lack more intricate features of Desktop Publishing software, such as a high-quality hyphenation and justification algorithm as well as chainable text boxes that allow a paragraph to continue across multiple text areas not contained within the main flow of the document.

The academic sphere, especially engineering, natural sciences, and mathematics, has widely rejected these commercial offerings for setting documents and publications. Instead, LaTeX has become the widespread solution for the typesetting of scientific documents because it can appealingly set mathematical formulas. It is an extensible open-source platform that emerged from the sciences themselves. LaTeX is a text-based system, meaning that the user prepares a source file in the TeX language, which contains a mix of content and instructions to format that content. The file is then translated into some output format, most frequently PDF, by a compiler.

Text-based systems enable non-destructive workflows for layout and styling. In such systems, the user can specify the intent of a block of content separately from its appearance, removing the tight coupling between formatting and content. The user can change the style of any type of content block, even for a finished manuscript. They retain the flexibility to make local changes to the style of an element without interfering with the document's style rules. Text-based typesetting systems also easily allow creating, sharing, and editing style and layout libraries that groups of users can reuse across multiple documents. Organizations can leverage these shared styles to quickly and consistently apply their design system to all documents, or a scientist can adapt their paper to the required style of a publisher by changing a single line.

However, LaTeX is unpopular outside of academic circles. In the Enlyft 2022 desktop publishing market overview, TeX or related products receive not enough usage to be listed. Two factors alienate users from the software:

1. It requires a considerable amount of upfront technical learning to become productive with the system. Learning a code language, memorizing its—at time arcane—command names, and learning how to make sense of the error messages LaTeX emits takes time and effort. Although a commercial WYSIWYG system also requires training, it is easier to arrive at a basic document.
2. LaTeX processes changes slower than a WYSIWYG system. Once a user has made enough changes to their document to want to preview them, they have to run the LaTeX compiler to obtain a PDF or DVI file, which can be viewed either in an integrated IDE panel or an external viewer. These LaTeX compiler runs often take multiple seconds to complete. The experience of editing a document in a WYSIWYG system is quite different. Upon each keystroke or layout property change, the software instantly provides a refreshed preview of the changes, even if it might

not match the fidelity of the final output (for example, raster image resolution or glyph outline quality). LaTeX's sluggishness is particularly disruptive when trying to fine-tune some style or position parameters, where there is no clear "right value." Instead, the user must optically judge which set of parameters "looks just right." The slow operating speed is also a problem when editing a paragraph to fix bad linebreaks.

Typst is a text-based desktop publishing system that seeks to address both of these issues: Its markup language is intuitive without sacrificing flexibility and extensibility. It also makes recompilations near-instantaneous, enabling up-to-date previews after each keystroke. The Typst standard library provides more useful high-level primitives than that of LaTeX, obviating the need to load external packages for tasks like reference management and image handling. Common formatting tasks like adding headings and lists are supported through dedicated markup (similar in syntax to Markdown; Gruber 2004). The markup is designed to capture how people might format plain text files, contributing to good source readability. Typst also includes the capabilities of a traditional programming language, enabling powerful automation and custom styles. Finally, the primitive layout functions that ship with Typst seek to encapsulate common layout patterns like stacks or grids of content while remaining composable to realize more exotic layout ideas. As previously explored, low preview latency is another factor paramount to a good user experience. This thesis shall explore and contribute techniques that enable Typst to meet its performance goal.

The composition process of most documents starts with an empty page or a small template that even LaTeX can compile in a reasonable timeframe. Throughout writing and setting the document, a user will continue adding content, layout instructions, and styling to the document until the typesetting software lays out all desired content in the intended fashion. The text-based typesetting system must compile multiple source file versions of increasing complexity. However, only a few pieces of the source file change between each compiler run, most of it remaining the same. Nevertheless, a conventional batch compiler has to reprocess the whole file from scratch each time, even if many artifacts that the compiler creates internally during a run are identical to those created in previous runs. In typesetting systems, this is particularly expensive since tasks like font shaping, reordering for bidirectional text, and layout computations are taxing operations. The follwing sections explore these typesetting tasks and requirements in more detail, present the architecture and features of TeX and Typst. The motivation behind the development of Typst is presented throughout the chapter.

2.1 The Art of Typesetting

This section will review what is required for typesetting a page in an appealing way through a LaTeX lense: It will explain how LaTeX and TeX are structured and how they achieve various typesetting tasks or where they fall short. In this section, all information on TeX is from Donald Knuth's indispensable TeXbook (Knuth 1986) unless otherwise indicated.

2.1.1 Arranging Content on a Page

This section explores what tasks and challenges a typesetting system encounters when transforming content, e.g., glyphs and vector or raster graphics, and the specified user layout intent, the style, and positioning with which the user wants these objects to

appear on the document's pages, to a paginated output file.

Authors organize documents with subdivisions like paragraphs, figures, footnotes, tables, pull quotes, and more. These subdivisions dictate how their constituent parts get laid out and where page breaks are necessary. Typesetting systems often provide custom layout containers that can contain other content subdivisions. For example, a document may contain many paragraphs. The typesetting system bundles all paragraphs and other equivalent items (*block level elements*) into a vertical stack and inserts inter-paragraph spacing, adding page breaks between or inside of block level elements when necessary. The layouting system often inserts organizational layout controllers for the above purposes automatically. However, it may also choose to expose such primitives to the user so that they can take fine-grained control and create more intricate page layouts.

In many primitives, greedy first-fit algorithms where all children are laid out sequentially and only once are incapable of providing the desired aesthetic effects. For example, the user might want equally long columns in a two-column document. The layout system now has to know the dimensions of all containers to lay them out in a balanced way. If the system may break the containers to enhance the balance between columns further, the size of the fragments does not necessarily match the size of the whole. Therefore, multiple layout passes for a single element may be needed when allowing elements to break. Grid layouts are especially notorious in the complexity required to allocate each row and column the desired space.

Some content in a document, like citations, may require type to be set across multiple locations (in this case, within the paragraph that the literature was cited in and then in the reference listing). Unlike other layouting systems, like a web browser, unique problems arise because the individual layout target areas, the pages, have finite dimensions. However, there is a possibly infinite number of them. Consider a footnote: When a superscript number referencing a footnote appears within a paragraph, the typesetting system must guarantee that the corresponding footnote appears on the same page. Positioning the note could become a problem if it is larger than the remaining space on the page or if the line referencing the note would be the last line on the page if the footnote was not set. In these cases, the typesetting system has to decide if it wants to break the lines and paragraph anew or whether it wants to break the footnote across two pages. Conversely, suppose a browser sets a footnote. It has two options:

- Putting it at the end of a page or directly below the referencing paragraph, possibly at a smaller font size, and moving down the other paragraphs
- Extending the page after the last paragraph

Both are trivial because web pages typically grow vertically to fit their content. Even if the page was forbidden to grow beyond some dimensions, e.g., the user viewport, there is only a finite amount of space, so, subject to the stylesheet, the browser puts all items that fit on the page. The rest will either be omitted or overlaid.

Furthermore, a typesetting system has to provide capabilities to insert artifacts like page numbers that depend on the page counter or other data about the partially finished layout. If the system allows for it, these artifacts can, in turn, affect the page layouts and thus page breaks, possibly creating cyclical dependencies. The same applies to textual references to other pages within the same document. Tables of contents are a well-known example of this behavior. Their length can shift on which page a section ends up, changing the page information of their entries.

Within paragraphs, glyphs must be composed into words. Those words and other *inline level elements* (e.g., formulas, images) must then be split into separate lines. The algorithm for ragged margin line breaks is easy: Just fit words onto the line until adding the next word exceeds the maximum permissible line length. In many documents, such as scientific articles or newspapers, however, justified paragraphs are preferred: Within justified paragraphs, the whitespace between glyphs and words is stretched until the first and last glyph touch the edges of the line bounding box. It is desirable to keep this stretching minimal to preserve a natural text appearance (Felici 2012). The naïve algorithm considers all possible ways to break the paragraph and calculate the average word spacing deviation from the optimum. This algorithm would be exponential over the number of line break opportunities, thus introducing prohibitive computational complexity. Dynamic programming solves this problem. The commonly used algorithm in (Knuth and Plass 1981) also considers advanced parameters like keeping the stretching factor variations between neighboring lines minimal and minimizing subsequent hyphenating line breaks.

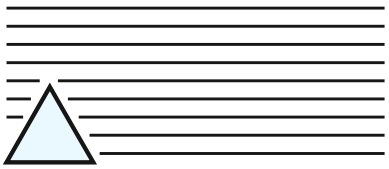


Figure 2: A graphic interrupts the lines in a paragraph. The lines adapt to the shape of the graphic.

Another challenge is the interruption of paragraphs by graphics that displace the text. While pages, text boxes, and paragraph bounding boxes are all rectangular, some graphics are not. A typesetter will sometimes place such graphics in a body of text such that the lines run right up to the graphic's contour (cf. Figure 2). In such cases, the typesetting system must locate the discontinuities in each line that collides with the graphic and calculate new line lengths, causing the lines in a paragraphs to be set into different measures (line widths). When combined with justification, colliders become a source of considerable complexity.

2.1.2 Modern Representations of Text

Donald Knuth designed TEX for systems with single-byte text encodings. The most popular such encoding is ASCII, which assigns each byte value between 0 and 127 a character. Some of these are control characters, like 10 (line feed), which creates a new line, or 7 (bell), which would make the terminal ring upon receiving the character from the mainframe. These characters are present due to historic typewriter-style terminals (The Kermit Project 2019), which are also the reason that separate from the line feed, there exist control characters for returning the carriage to the starting position and feeding a new page). All characters that do not control the terminal unambiguously map to a glyph, so 65 would always yield the uppercase A glyph. TEX's fonts contain up to 256 glyphs, enabling support for encoding extensions ("code pages") like Latin-1, assigning the remaining values in the 128-255 range to other characters like Ø. Depending on which other characters were needed, the user would swap out the code page for the second half of the text bytes' value range, both in TEX and in their editor. TEX switches to a different font with Greek and mathematical symbols when setting

mathematics. Classic TEX, however, only can use 16 fonts in one document.

There are a few exceptions to the byte-glyph mapping. A sequence of the bytes for 'f' and 'i' respectively will produce a single 'fi' *ligature* glyph, taking up one slot in the font, but these were mostly decorative. These ligature rules are hardcoded into the TEX system for specific character pairs and fonts. TEX can easily accommodate English texts and texts written in most European languages with this font and encoding system. However, it runs into problems with scripts like Simplified Chinese, containing 8105 characters, or the Japanese Kanji script, containing over 13,000 characters (not to mention that many texts will also use other Japanese scripts like Hiragana and Katakana) (Beebe 2004). Authors of documents in these languages often use special packages or modifications of TEX that ship with several fonts. They are exchanged on the fly, depending on the character to be printed (Lagally 2004). These non-standard extensions render document exchange more complex and slow the TEX compiler down.

The Unicode standard was introduced in 1991 to standardize a way to encode every character in every language within a single, consistent scheme (Aliprand 2011). Unicode introduces the concept of code points: The standard assigns each character a code point, of which there is a total of 1,114,112. Representing this naïvely in a file would require each character to be four bytes long, quadrupling file sizes. Instead, the most popular encoding for Unicode is UTF-8, which encodes code points with one to four bytes. For each byte, the most significant bit decides if it terminates the code point encoding and the remaining seven bits carry data (Yergeau 2003). The first 127 characters in this encoding map to their code points; hence compatibility between ASCII and UTF-8 is retained.

Along with Unicode, OpenType fonts were introduced. An OpenType font contains glyphs for a selection of Unicode codepoints. However, the mapping from code points to glyphs is not 1-to-1 since the font can produce a single glyph for the combination of multiple code points. Even when a single code point only results in one glyph, this glyph might depend on the surrounding code points in the text string. As an example, consider the emoji from before, where a sequence of two code points will produce a single glyph, differing from the glyph that either code point would have produced on its own. In English, the number of such n-m relations between code points and glyphs is manageable. However, languages like Arabic use different letterforms depending on where in the word a letter stands. Thus, many code points correspond to three or more glyphs depending on their position in the word.

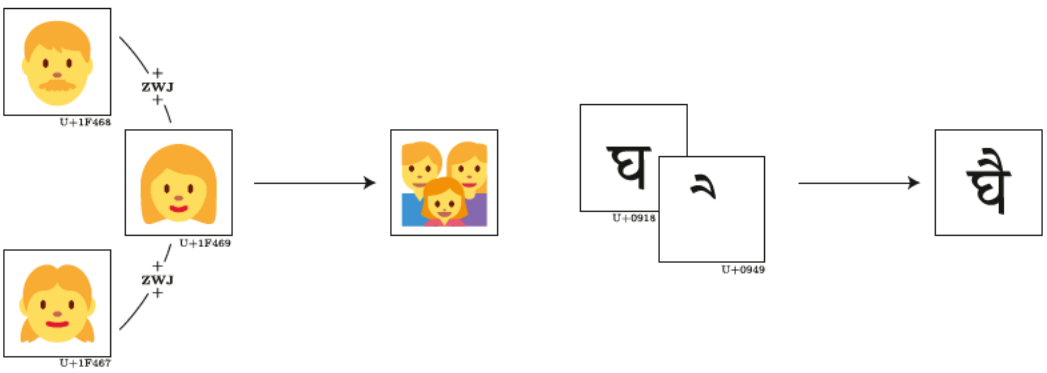


Figure 3: Composing Emoji and Devanagari characters with multiple Unicode codepoints.

The question now becomes how to select the correct glyph for a code point, depending on its context. Due to the high number of fonts and the structural freedoms afforded to them by OpenType and the more than 120 scripts supported by Unicode, it is not viable for the typesetting system to contain rules to deal with every conceivable font and script. Instead, a font will encode information about glyph selection within internal tables (the `GSUB` or `morx` tables play an important role here). A program called "shaper" is processing this information. The shaper ingests the font and a text string. It returns the appropriate glyphs for the font and how far to advance the drawing cursor. For example, the Thai character ก็ is created by printing the combining accent glyph ◌็ and then not advancing the drawing cursor to print the ก glyph. Because the shaper decides where each glyph starts, it can also adjust the spacing between individual character pairs. For example, the sequence 'To' has the 'o' closer to the capital letter than within the sequence 'Mo.' Print designers call this adjustment of character positions *kerning.* Kerning keeps the distances optically equal. Fonts may supply the shaper with tables containing kerning glyph pairs. The shaper will also mark safe word-break opportunities.

Our Thai character is a *grapheme cluster,* another example would be the pride flag emoji 🏳️‍🌈 consisting of the code points for the blank flag emoji, the zero-width joiner (indicating that the two adjacent characters ought to be joined), and the rainbow emoji. Of course, breaking up this sequence of code points with, e.g., a line break is not permissible, whereas breaking up the 'fi' ligature would be okay in principle. Unicode defines a grapheme cluster as a sequence of code points that produce a single user-perceived character, differentiating it from ligatures, for which a sequence of code points produces a single glyph that consists of multiple user-perceived characters. A modern typesetter, browser, or text processor would invoke a shaper with its code points to determine the correct glyphs. If it needs to break up a word after it determines that the returned glyphs are too wide for the current line, it must select a legal place to split the shaping run (e.g., not within grapheme clusters) and then rerun the shaper for both halves. Windows and macOS provide shapers (within DirectWrite and Core Text) for native applications on their operating systems that co-evolved with the respective TrueType and OpenType font standards. The most comprehensive open-source shaper implementation is HarfBuzz, used in many Linux distributions (including Android), LibreOffice, and the Chrome browser.

Another consideration for modern typesetting systems implementing the Unicode standard is the concept of bidirectionality. Some scripts, like Arab and Hebrew, are written from right to left on a page. However, within a Unicode-encoded file, the code points for their characters appear in the logical reading order. Such files may also contain an arbitrary mix of left-to-right and right-to-left code points, often used within Arab documents for numbers or foreign names. Any system that renders such text files can thus not simply start at either side of the viewport and place shaped runs next to each other. Instead, it must identify *directional runs of text* that only contain text running in either direction and then shape and place them separately. For this purpose, Unicode standardized a bidirectionality algorithm, defining how different classes of code points affect the order of glyph drawing (Atkin and Stansifer 2000). Some characters, like braces, even are neutral with respect to directionality. Because the brace code points are specified as opening and closing brace, the shaper needs to select the appropriate glyph for the directionality of the surrounding script (i.e., flip the opening and closing glyph). For this and other reasons, the shaper must also be aware of text directionality.

Because the order of characters in the final document depends on where the line breaks occur, bidirectional reordering is a two-step process: First, the *embedding levels* are determined. Each nested level constitutes a directionality change. Then, after the line breaks are determined, the actual glyph order is determined based on these levels.

2.2 The Heart of TEX

In order to give the reader the background necessary to examine the trade-offs involved when designing a typesetting system, this section explains the inner workings of Donald Knuth's TEX.

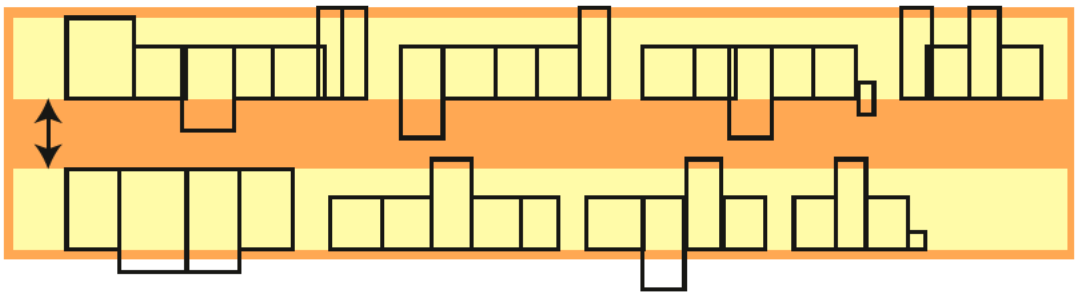


Figure 4: Anatomy of a paragraph in TEX. The black boxes represent (letter) boxes, the light yellow rectangles are horizontal lists, the orange rectangle is a vertical list, and the arrow denotes inter-line glue.

TEX is a macro-based typesetting system. It works by assembling the document content into boxes of fixed size and the space in-between into glue, which can stretch and shrink beyond its natural value to benefit the document's visual appearance. Fundamentally, TEX will map the source text to glyphs as described in the previous section. It then creates tightly fitting boxes for each glyph and annotates them with the position of the baseline (the vertical position on which a glyph aligns with the others). The space characters within a paragraph are transformed into inter-word glue. TEX then collects the boxes, glue, and other artifacts into a *horizontal* list that may contain *penalties* describing how bad (or desirable) a line break at a particular location. Breaks are allowed between many horizontal list items, but some sequences of more obscure entries like a math-mode-enabling marker followed by a glue disallow them. *Discretionary breaks* are possible linebreak opportunities within a word, commonly inserting a hyphen. They contain three boxes (two to place before and behind a break if the break occurred and one to show otherwise; this deals with words that change spelling when hyphenated). TEX executes the Knuth-Plass algorithm (Knuth and Plass 1981) to determine the line breaks for the break opportunities within the horizontal list. The horizontal list items are then embedded into a set of horizontal boxes, one for each line. The horizontal box aligns all of its items on their baseline and shows them from left to right. Line and paragraph spacing is inserted as glue between the horizontal boxes.

Pagination then works according to the same principle: TEX puts all horizontal boxes and glue in a *vertical list* for which the same criteria for valid page breaks apply. The list is then separated into various pages (vertical boxes) using an adaption of the line-breaking algorithm. Objects like floating figures and footnotes are dealt

with through a unique mechanism called 'insertion.' Insertions may occur within the vertical list but not in nested boxes or other insertions. The insertion may appear at the top, bottom, or center of a page. Some insertions, like footnotes' first lines, must appear on the same page where their sibling elements reside, but most can also move to a queue to appear on the following pages. When processing the vertical list, TEX considers the desirability of a line break for every line and the space that the insertions on the page's lines and from the queue would take up. The algorithm locally searches for the best place to break the page and moves a chunk of the vertical list into a vertical box. Headers and page numbers are then inserted in post-processing routines and superimposed onto the finished page, meaning that they cannot influence the page content (e.g., with their height).

The insertion example illustrates a TEX design principle: No backtracking or global optimization is allowed apart from line-breaking and pagination. Instead, commands are designed to take effect at the first possible location after their declaration, amounting to a greedy (Mittelbach 2014) placement of most elements. That also means that a table of contents cannot look through the document to collect section headings or that references cannot look back at the referenced location to determine the section number. Instead, TEX macros write auxiliary files during the compilation process that contain information that needs to be available when the commands take effect. For example, TEX will set a table of contents as an empty table during the first compilation and write all start pages of the sections into an `.aux` file. In the second compilation, the table of contents macro can reference this `.aux` file to extract the right content. The rules for automatically managed bibliographies are typically not implemented as TEX macros. Instead, a companion macro will write information about the used citations to the compilation's `.aux` file. An external program like BibLATEX processes this file to emit a `.bbl` file containing the right macros to cite in the right style and populate the references section. This two-step process is also why a typical document with citations has a complex compilation cycle: First, the user has to run TEX to obtain an `.aux` file. Then, the bibliography management software is run to emit the `.bbl` file (Kime and Charette 2022). TEX then needs to run again with the information from both files to produce all references and reference tables. If, however, the table of content now occupies two pages, each reference becomes off by one page. TEX finally has to run a third time to adapt to the new document length and correct the references.

Macros and primitive commands control the TEX software. Its compiler interleaves tokenization, parsing, macro expansion, and typesetting. The file is simultaneously read and typeset, as Figure 5 shows. Every character belongs to a character class

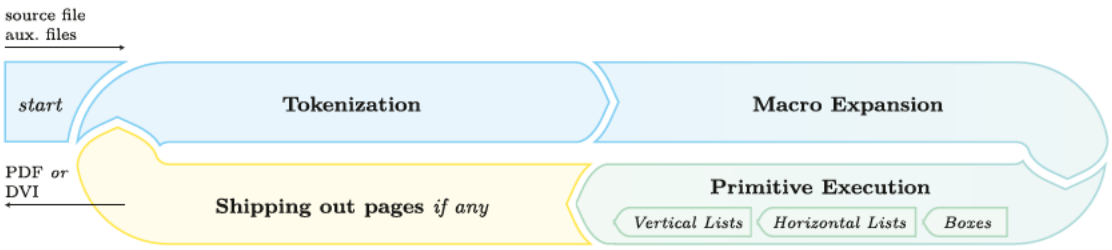


Figure 5: The TEX compilation process according to Abrahams, Hargreaves, and Berry 1990: Parsing, macro expansion and layout tasks are interleaved and new input from the source file is continually added.

(i.e., opening or closing braces, spaces, letters). The character class some byte belongs to can be changed within the document source code and takes immediate effect afterward. Because any given byte thus does not correspond to a single token, the tokenization has to run in parallel with the interpretation of the document primitives. Macros allow defining names for a sequence of primitive instructions or other macros, which then are *expanded* into their definitions when found. Their arguments need not appear in a group following the invocation. Instead, the macro can specify what sequence of tokens it expects as an argument. Macros cannot be assigned a value at runtime, only a set of tokens predefined in the source. This limitation makes them inappropriate as variables. Hence, TEX provides a set of registers for each data type. Macros can ask TEX to allocate a free register to them and assign it to a name such that no collisions occur.

TEX users rarely just use the primitive commands because they operate at a very low level. It is possible to define a custom set of macros for layout operations commonly found within one's document from scratch. However, most users use a predefined set of macros (maybe with some custom user-defined macros on top) for their documents. These macro sets are called TEX formats. All installations of TEX come with the 'plain TEX format' by Knuth. The format's name is a bit confusing since it provides additional macros on top of what TEX supports as a primitive operation. Plain TEX includes macros that control line and page breaks, insert rulers and spacing, produce spacing, change the font, and set special and accented characters. Leslie Lamport has published the popular LATEX format with macros for lists, additional Math symbols and constructs, figures, tables, tables of content, section headings, and a package system. LATEX intends to obviate the document source file from specifying how exactly the document should look. Instead, a user of the LATEX format must specify a *document class,* characterizing the kind of document they are setting and then annotating it with semantical macros such as `\section` to indicate section headings. LATEX will then take care that they look appropriate for the document class. Lamport therefore LATEX characterizes as a document preparation system, that takes care of the typographical details for the user (Lamport 1986). TEX formats are typically loaded as binary `fmt` files that the `inittex` variant of the compiler can create to save time for string operations. Most modern TEX distributions provide compiler commands that preload LATEX, but calling `tex --fmt=latex file.tex` is equivalent to calling `latex file.tex`, in both cases, `file.tex` is compiled with the `latex.fmt` format file being preloaded (Knuth, Trickey, et al. 2020). Modern applications of the LATEX format often depend on a great number of packages, like, for example, for graphics and bibliography management. However, the LATEX package system has the limitation that a package cannot be loaded as a precompiled `fmt` file because the source specifies which packages it uses in-line. LATEX instead loads packages as source files and inserts them into the user source, interpreted as the other user code.

Donald Knuth first released a TEX compiler in 1978, the first compiler implementing TEX as known today was published in 1986. Today, a TEX distribution ships as a software package that contains various TEX formats, documentation, (LA)TEX packages, and compilers. Alternative TEX compilers seek to implement the typesetting language while adding features or addressing some of its shortcomings. ε-TEX is a TEX compiler from 1996 that includes some helpers for macro developers, but most importantly, it increases the maximum register count for each datatype from 256 to 32768 (Breitenlohner 1998). At the time, TEX users first started to run into problems of being limited by the traditional TEX compiler's register count. Another compiler that was

built on ε-TEX' foundation is pdfTEX. This compiler introduced two significant new features: Its output is in PDF format instead of Knuth's DVI format. PDFs advantage is native support of raster graphics as well as features such as hyperlinks (Thành et al. 2021). Additionally, pdfTEX introduced new microtypographical features that allow characters to extend beyond the margins of their column to make it appear visually straight. Most prominently, the hyphen will protrude beyond the column width because otherwise, the line it appears on will not look long enough (Thành 2000). XHTEX is a compiler originally developed for macOS and then ported to other platforms. It natively processes Unicode-encoded source files and uses the operating system's font and shaping capabilities (Goossens 2010). XHTEX also lifts the restrictions on the number of loaded fonts and characters. Another similar project is LuaTEX which aims to make the TEX compiler completely scriptable using the Lua language. It, too, is designed with Unicode in mind (LuaTEX development team 2022). Its shaper (the ConTEXt OpenType engine; Eigner 2017) is implemented from scratch in Lua and supports OpenType Maths fonts for mathematical typesetting, but sometimes lacks the fidelity the native shaper provides to XHTEX, especially for bidirectional text. LuaTEX is in active development.

2.3 Typst

2.3.1 Language

Typst is a new in-development text-based desktop publishing system controlled by the eponymous typesetting language. This section explains essential concepts and the design rationale. A more comprehensive explanation of the language can be found in Mädje n.d.. The examples in this section show the Typst source code on the left and the resulting compilation output on the right.

The Typst language is Turing-complete, dynamically typed, and not macro-based. It provides capabilities for user-defined, possibly variadic functions, variables, arrays, dictionaries (hash maps), and a module system. A Typst file specifies a document that the compiler should set. Typst combines typical markup and scripting/programming elements into a single context-sensitive grammar. On the one hand, the user can type their text directly into the file and use special markup syntax to produce common typesetting patterns. On the other hand, they can access the programming features by opening a code block delimited by curly braces. Typst's programming features can be used inside these code blocks to insert content and modify the document's appearance. The code block swaps the markup syntax for a more traditional programming syntax similar to many imperative languages. Here, the user may define functions and variables as well as use constructs like expressions, conditionals and loops.

The Typst language is designed to accommodate desktop publishing needs both in syntax and semantics. It provides specialized data types like absolute and relative lengths and combinations thereof in various customary units. The most notable type is *content,* a combination of text and markup. It appears top-level in the source file

```
Hello World!

1 + 2 = {1 + 2}
```

Hello World!

$1 + 2 = 3$

Listing 1: Text and scripting in Typst.

or as content blocks within a code block. A template can be typeset, inserted into another template, or modified by functions. Content can be stored in a variable such that code can insert it into other content or reuse it. Content can contain styling and formatting directives. Typst will derive all style properties that the content did not set from the style of the insertion site such that the content's formatting always matches the surrounding content. Within a content block literal, enclosed in square brackets, the markup syntax is active. There is a convenient syntax shortcut to access single programming operations within a template. The `#` prefix allows performing a single code operation, such as a function call, a variable declaration with `let`, or an `if`-conditional.

```
#let a = [This too]

This is content and #a!
```

This is content and This too!

Listing 2: Storing and using content in variables.

The Typst compiler ships with a standard set of constants and functions, and they allow access to its internal typographic and layout capabilities or act as convenient helpers. This standard library is also implemented in Rust and is loaded for all documents because, otherwise, most of Typst's features are not usable.

The markup syntax is very similar to languages like Markdown and AsciiDoc and shares their aim of making the source file as readable and well-formatted as possible, resembling the output's appearance. The features accessible through specialized markup are common in many documents or frequently used in some documents, and all functionality provided by markup is also available through library functions. As a markup example, the user may set a bullet point list by putting a hyphen at the start of a line, typing a space, and then typing the content to appear for that list item. Second-level list items can then be created by, on the following line, indenting the next hyphen with a tab or spaces. *Stars* surround **strong, boldfaced text** and _underscores_ create *emphasized, italicized text.* Some sequences produce characters that are frequently used in published texts but do not appear on keyboards, such as the em-dash (—), settable by typing three hyphens in a row.

Of course, most users will want to override the default styles for built-in elements like enumerations. The standard library contains functions for each markup item, and the markup invokes the corresponding functions for each item, which are then set onto the page. The users can manipulate the appearance of each function through either its arguments that apply to the current invocation or its associated style properties with the `set` keyword. These styles take effect from that point in the content onwards (cf. Listing 3).

```
#set enum(label: "a)")

Q: What is the capital of
California?

1. San Francisco
2. Sacramento
3. Los Angeles
4. Fresno
```

Q: What is the capital of California?

a) San Francisco
b) Sacramento
c) Los Angeles
d) Fresno

Listing 3: Adjustment of enumeration labels with a set rule.

Sometimes, the customization afforded by the function styles is not sufficient to realize the intended layout. Users who want to customize a heading more than possible through its properties could define a custom function (cf. Listing 4) that styles a template argument accordingly. This solution, however, would have two drawbacks: The definition of a custom function removes the possibility of using markup to create headings and instead forces always calling the function manually. Also, Typst is unaware that text styled through this function is semantically a heading, and it cannot mark it as such in the output or search for it at other places in the document. This lack of information breaks automatically generated tables of content, the outline panel of a PDF viewer, which allows the user to skip to a section directly, reduces the fidelity of screen readers and other accessibility helpers, and general machine readability such as by web spiders.

```
#let myheading(
  level: 1,
  body,
) = block({
  set text(
    size: 11pt + (6-level)*1pt,
    weight: "bold",
  )
  if level == 1 {
    underline(body)
  } else {
    body
  }
})

#myheading[Background]
Lorem ipsum

#myheading(level: 2)[Tex]
Abc
```

Background

Lorem ipsum

Tex

Abc

Listing 4: Definition of a custom heading function[1].

[1] Since function calls with one template argument are common, Typst provides a special syntax for them which allows omitting a pair of parentheses: `box[Hello World]` desugars to `box([Hello World])`.

Some functions carry semantic meaning that the user cannot otherwise create. However, parts of the implementation of these functions can be overridden, preserving the original semantics. Users can write a `show` rule that dictates how the function will transform its arguments into content. The show rule binds a dictionary with information about the invocation and the arguments to a variable available to the custom implementation (cf. Listing 5).

For example, the dictionary will contain an array of content for the list function, one entry for each list item. The implementation can then process the array arbitrarily to produce more sophisticated layouts. In Listing 6, lists produce grids, and here, they are populated with images.

```
#show it: heading as {
  set text(
    size:
      11pt + (6-it.level)*1pt,
    weight: "bold",
  )
  if it.level == 1 {
    underline(it.body)
  } else {
    it.body
  }
}

= Background
Lorem ipsum

== Tex
Abc
```

Background

Lorem ipsum

Tex

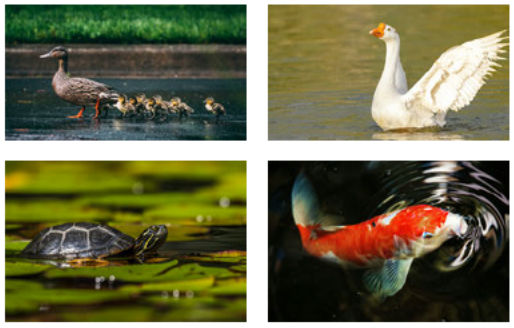
Abc

Listing 5: Customization of the built-in headings with show rules.

Variable declarations and changes made by `set` and `show` are scoped, and both content and code blocks are scoping. A function cannot "return" styles for the calling content. Instead, they can only style their arguments with their interior styles. For this purpose, Typst provides the `wrap` keyword. A `wrap` binds all content below it into a variable which can be passed in a function call. This allows organizations to provide style packages that set up everything in the right way so that markup will always produce appropriately styled elements, making it easy to consistently enforce a corporate visual identity or a style of a scientific journal. Listing 7 shows how `wrap` can apply an imported conference style.

```
#show it: list as grid(
  // create two equally large
  // columns that take up all
  // the available space
  columns: (1fr, 1fr),
  gutter: 8pt,
  // insert all list items,
  // new rows are added
  // automatically.
  ..it.items
)

- #image("duck.jpg")
- #image("goose.jpg")
- #image("turtle.jpg")
- #image("carp.jpg")
```



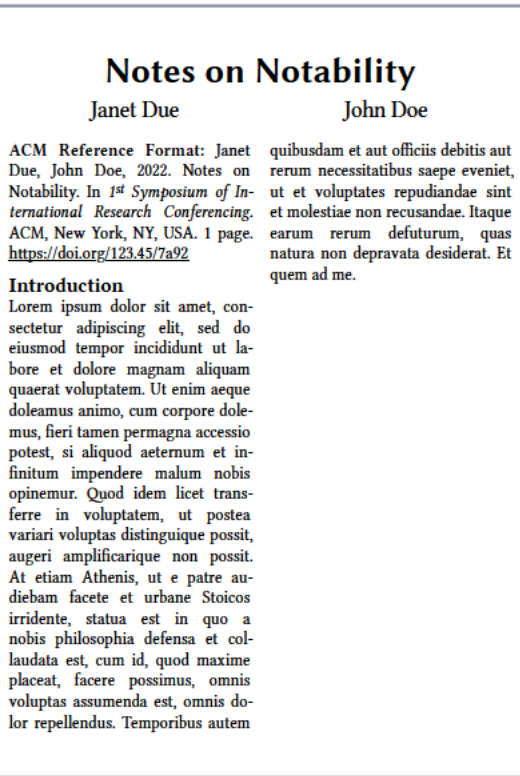
Listing 6: Redefining the built-in list as a grid with show rules.

```
#import conference from "acm"

// Wrap the remaining paper
// in the function.
#wrap doc in conference(
  title: [Notes on
Notability],
  authors: ([Janet Due], [John
Doe]),
  // ... other properties
  venue: [1#super[st]
Symposium of International
Research Conferencing],
  // The remainder of the
  // paper is passed here.
  doc
)

= Introduction

#lorem(130)
```



Listing 7: Using an imported style with `wrap`.

2.3.2 Compilation process

Figure 6 shows the four stages of the Typst compilation process and what representations they result in.

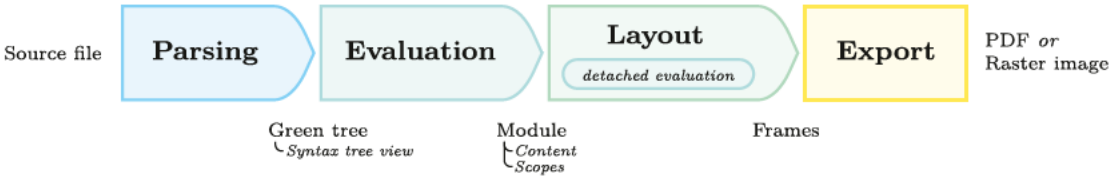


Figure 6: Stages of the Typst compilation process. Like programming language compilers and interpreters, Typst parses and interprets a file before layouting it.

The compiler proceeds sequentially through these stages in simple documents. However, it may invoke earlier stages at some point. For example, when resolving imports during evaluation, the Typst compiler has to fetch, parse, and evaluate the imported Typst files to proceed using their module. Similarly, the content of show rules is executed during the layout phase. Here, the evaluation does not result in a full module. Instead, it only returns the function result obtained with a detached evaluation context without I/O capabilities.

Our tokenizer first transforms a plain string into an iterator of tokens. During this process, the tokenizer performs some lookahead. We then use these tokens in our handwritten recursive descent parser to parse these into nodes. A node could be a space, text, expressions, or markup; the parser does not form paragraphs. The parser places code constructs like variable declarations in `Expr` (if using the `#` shorthand) or in `CodeBlock` nodes. Parsing variable bindings as expressions might be unusual, but Typst interprets every "scripting" construct as an expression. Most of them implicitly return `none`.

In the evaluation phase, the nodes are associated with their respective values. Thus all expressions, loops, conditionals, and functions are executed. A top-level scope containing the standard library is injected here. Code blocks and content blocks add their scopes during the evaluation. Operations like calculations or fetch requests from the disk get resolved at this stage. However, other information like where on a page the content for a node gets printed on is still undetermined. Functions can yield templates that exhibit custom behavior by containing native code. The evaluation runs once for every source file, and its result is a *module*. Modules contain the global scope of a source file, with all importable functions and variables as well as the file's top-level content. A module represents content as a tree. Each content node is one of the following:

- A leaf (e.g., linebreak, text)
- Injection of some styles with a single child (produced by a set- or show-rule)
- A sequence node with multiple children

The layouting phase converts content into a tree of layoutable nodes (layout node). For these, it computes the physical locations of items, performs font lookup and fallback, shapes text runs, and calculates line breaks. Both processes are interleaved. It yields a list of `Frames`, where each frame is a container of a fixed size containing atomic visual elements like positioned glyphs, shapes, and pictures. The layouting phase traverses

the content tree and collects its items in page runs (one or more pages not interrupted by an explicit page break), each containing a top-level flow. Only now, the flow collects the text leaves of the content tree into paragraphs. The content tree may also contain block-level (siblings of the paragraphs) and inline-level layout nodes (children of the paragraphs), which the flow needs to incorporate into its layout tree. These elements originate from calls to functions like `stack` or `grid` which explicitly produce a layout node. Each layout node has to provide a function that returns Frames given a context and the available space. The available space can either be a single rectangle or a sequence of rectangles of different dimensions, possibly on different pages. Most layout nodes can contain children, which they recursively layout. They then proceed to layout themselves.

Then, finally, these frames are passed to a format-specific backend, currently for PDF and raster images. These exporters write their respective formats to match the layout described by the frame list. Backends for formats like HTML/CSS could be added but might skip the layout phase.

Although element positions are unknown in the Evaluation phase, users still ought to have the possibility to retrieve information like the current page number programmatically. Locators serve this purpose: A locator can be placed in content (including the top-level document content) and referenced through a variable. It will be layouted with the content around it. The reference answers queries for its absolute position on a page (or just the number of the page it is on). The position can then be used to print a page number or perform some more advanced operation. Locators let users run code based on data that changes where the locator gets layouted, creating a cyclical dependency. Typst resolves the cycle by looping the Evaluation and Layouting phases until convergence. Locators will be initialized with a default value which is updated with the new layout position after each compilation run. Typst recompiles the document until no further changes occur or until it exceeds its recompilation budget, preventing endless loops due to non-convergence.

2.3.3 Platforms

Outside of the scope of this thesis, we develop the Typst compiler. It is implemented in Rust, whose is based on LLVM, enabling it to emit performant native code for all common processor architectures and operating systems, making creating a command-line app for the compiler easy. However, the main focus of the Typst project is to enable the user to set documents in their browser without installing other software. There, the Typst compiler can tightly couple with an editor to form a fully integrated typesetting web application. WebAssembly allows this. It is a technology that enables browsers to load and execute platform-independent assembly code not managed by the JavaScript virtual machine. WebAssembly (WASM) applications can reach near-native performance. When the Typst compiler is compiled to WASM, we can ship it in a web app containing a rich code editor and a dedicated export stage that renders Typst's frame output to an HTML canvas.

The tight integration of the editor and compiler shall provide a better editing experience. Notably, the editor can notify the compiler of which changes the user has applied to the document instead of passing the complete source code between threads. Conversely, the compiler can send the editor information about the AST and the compilation, enabling rich autocompletion experiences. Running the compiler in the browser prevents costly server-client roundtrips, enabling to trigger a document recompilation and preview after each keystroke without worrying about server load and

scalability issues. Another advantage is that the application has the offline capabilities expected of a Progressive Web Application: The browser downloads all necessary code and stores it with a web worker. The app only needs its network connection to sync with the server on every subsequent opening, not to compile a document. This architecture allows us to aim for fully interactive recompilation and preview, targeting the industry standard 30 and 60fps framerates.

3 Related Work

In this section, we give an overview of the state of the art in both incremental parsing and layouting. We start with the former, enabling the reader to better gauge this thesis' contribution in the context of the state of the art.

3.1 Parsing

In 1965, Donald Knuth invented LR(k) parsing. In Knuth 1965, he not only stipulated the LR parsing scheme but also defined the class of LR(k) grammars as the set of grammars being recognizable using this scheme. LR, which stands for Left to Right, is a bottom-up parsing scheme. Its name indicates that it builds the parse tree from the terminals upwards. An LR parser consists of a stack that contains the current terminals and non-terminals that might eventually be siblings in a parse tree and a cursor into the file that is being processed. The top-most element on the parsing stack encodes the parser's current state.

For each parsing step, the parser decides between two possible procedures based on the contents of the stack and the next k tokens starting at the cursor:

- Shift the next terminal token onto the stack and advance the cursor.
- Reduce the last tokens on the stack into another non-terminal, thus building the parse tree.

A parse table that indicates the following action given the current stack item and lookahead tokens facilitates this decision. Such tables are often automatically derived from grammars stated in a Backus-Naur-Form-like format by parser generators like `bison` (Aycock 2001).

The lookahead parameter k defines how expressive a grammar can be. For the grammar author, it is often easier to work with a large lookahead than a small one. In terms of power, however, there is just the distinction of languages with $k = 0$ and $k = 1$ because grammars requiring greater lookahead can be transformed into LR(1) grammars as proved in Sippu and Soisalon-Soininen 1990. There is a LR(1) grammar for each deterministic context-free language (Knuth 1965) which are a subset of all context-free languages (Hopcroft, Motwani, and Ullman 2006).

The GLR extension of the LR approach allows multiple actions for a state and lookahead combination in the transition table, which are all executed in parallel. Only one of these parse tree forks has to lead to a valid parse for the parser to accept the input. By extending LR parsing with this non-deterministic behavior, all context-free languages can be recognized (Tomita 1991).

Like many bottom-up approaches, LR(k) parsing is well-suited for incrementally adapting the parse tree to the user's changes to the source file. Because only the stack and the lookahead define the compiler's state, the source string merely needs to be reparsed from the moment the lookahead contains the first character of the change. The stack contents can quickly be recovered from the parse tree. The challenge is to determine when to stop the reparsing, i.e., when the old and the new parse tree have converged again behind the edit site.

In Ghezzi and Mandrioli 1979, the authors propose such an algorithm for LR(0) parsers: The parser annotates its tree with "threading" pointers with which each node points to the node that would be its predecessor in depth-first search. These pointers allow easy stack recovery: Once a node is identified as the first leaf containing the change, the stack can be filled by walking backward from its threading pointer and

queuing each encountered node. In this easily implementable procedure, the source must be parsed from the first changed terminal to the very end.

They introduce a second, *symmertrical* algorithm, require the grammar to be $LR \wedge RL$ and add another set of threading pointers to limit the extent of the reparse. This time, the threading pointers point to the successor of a node in depth-first search. Using the grammar's properties and the new set of pointers, the parser can immediately stop upon convergence and replace the relevant subtrees.

The main limits of this method are the requirement that the used grammar be $LR \wedge RL$ without lookahead. Many common programming languages cannot be expressed under these restrictions. This limitation is lifted in Jalili and Gallier 1982, which provides a procedure for LR(k) parsers where each parse tree node also contains the state the parser was in when it was added. Checking for convergence with the old tree after the change becomes trivial: The parser simply compares its stack and lookahead with the saved one. The problem here is that the saved state is not necessarily the only one permissible to reuse the following tree and the extra storage needed in each parse tree node.

Wagner tried to overcome these deficiencies in his thesis (Wagner 1998) by providing an LR(1)-based reparsing algorithm that is optimal with regards to node reuse. His algorithm "breaks down" the rightmost path of the sub-parse-tree of the change if necessary to perform the correct reductions and appropriately react to the lookahead.

Brunsfeld's Tree-Sitter (Brunsfeld 2018; Brunsfeld 2021) parser generator builds on these ideas by offering an incremental GLR parser. He specially built it for good error recovery and syntax highlighting. GitHub deployed Tree-Sitter at scale. The error recovery is accomplished by selecting the parse tree with the smallest error from the GLR parse forest. It operates by first updating the nodes in the original parse tree with the edit range in the source file. Then, the parser reconstructs its state ahead of the edit and processes the new terminals while reusing the parses of unchanged non-terminals from the old tree. Because the first step requires traversing and potentially changing the whole retained parse tree, the approach is not sub-linear for the original input.

All of the above schemes integrate with and require the usage of a parser generator and are thus very easy to use for new grammars (as long as the underlying parser generator supports them). This requirement, however, also imposes the limits of parser generators: Bad error messages, the ability to integrate with custom data structures, and the possibility to add "hacks" to recognize constructs that are not recognizable in the language/grammar class of the parser (Brink and Maniero 2010; tyoverby 2017). Schemes to overcome these weaknesses exist, for example, the possibility of generating error messages based on the whole parse tree using a database of examples (Jeffery 2003). However, they are problematic as they add extra components and can only approach the quality of tooling and usability of tools specifically designed for a language.

Because of this, many widespread programming language compilers like V8 (JavaScript), clang (C), Golang, and C# use handwritten recursive descent parsers (Eaton 2021). These top-down parsers are more difficult to "incrementalize" because they implicitly encode their state and pending reductions on the call stack, possibly in addition to the parsing functions' shared mutable state.

Nevertheless, there have been prior attempts to add incremental parsing to recursive descent parsers. The earliest such algorithm is defined for LL(1) grammars, a subset of LR(1) grammars (Rosenkrantz and Stearns 1970). It needs an additional

entry point for each production to start the reparse of a subtree in such a function and then uses the properties of LL(1) grammars to guarantee that the subtree associated with the edit is valid and can replace the prior corresponding subtree. However, many real-world languages cannot make use of this algorithm because they require productions that cannot occur in LL(1) grammars (Rossum, Galindo, and Nikolaou 2020). In comparison with Rosenkrantz and Stearns work, the incremental parser proposed by us is more general since it can accomodate not only LL(1) and LR(1) grammars, but also some context-sensitive grammars.

Some modern compilers, like C#, Swift, and the Rust developer tool `rust-analyzer` have instead leveraged the Red-Green Trees used in this thesis to achieve incremental parsing. The Roslyn compiler for C# has implemented incremental parsing for their recursive descent parser (cwzwarich and Lippert 2020) on the basis of a dissertation on one of their team members (Gafter 1990) on bottom-up parsers.

To the best of our knowledge, there is no detailed public description of Roslyn's incremental parsing process or its rationale. However, the basic procedure can be observed in the public source code (Parsons et al. 2022). Like this thesis, they associate source ranges with the nodes in the concrete syntax tree that allow mapping nodes from the previous tree to their changed counterparts. The range of the considered leaf nodes is then expanded, and the tokenizer is run. Once the changed and the original text tokens have converged again, a final set of changed leaf nodes in the CST can be collected. The parsing then restarts from some shared parent node, and the resulting concrete syntax subtree is patched into the existing tree. For this reason, their parser has entry points for many different language constructs.

They do not describe how this procedure guarantees correct results, given the properties of the C# grammar and the implementation.

The Rust development tool `rust-analyzer` contains a parser implementation distinct from the language's compiler, `rustc`. It is specially optimized for syntax highlighting, error resilience, and automatic refactorings in IDEs. They use a restricted reparsing scheme where they either reparse a single token if it contains the change or the enveloping code block (block wrapped by curly braces; Kladov, Tanaka, et al. 2021). The single token reparse can only occur for strings, whitespace, comments, and identifiers. The token's type must be identical before and after the reparse (Kladov, Singh, et al. 2022). This limited approach is similar to how this thesis' algorithm treats Typst's code mode. The limitation to blocks and specific tokens keeps the code dealing with the particularities of a grammar to a minimum and makes the approach more easily verifiable.

Another more formalism-driven approach is Packrat parsing. It is a parsing algorithm for Parsing Expression Grammars (PEGs). PEGs are a formal framework to provide grammars for top-down parsing (Ford 2004). They eschew the ambiguities that can arise in normal context-free grammars by providing operators that, if there is a choice between two possibilities, will always prefer the first over the second. Furthermore, they remove the need for separate regular expressions to recognize individual tokens. Due to this and the structure of the formalism, they are better suited to express nested structures. Languages that can be encoded in PEG grammars are a superset of the languages in LR(k) (Ford 2004). PEG has since gained traction in industry applications. Most notably, Python has switched to a PEG parser, in part because of the constraints imposed by their previous LL(1) grammar (Rossum, Galindo, and Nikolaou 2020).

Packrat parsing is a method for lowering the asymptotic runtime complexity of PEG parsers to linear time. A packrat parser is a backtracking recursive descent parser that, for each position in the source code, memoizes the results of calls to functions of the parser that accept various sub-expressions of the language into a table. When two possibilities of how an expression may be composed start with the same sub-expression, the sub-expression is only parsed once, even if the parser has to try and backtrack from the first alternative (Ford 2004). This memoization strategy is valid because "[the] memoization system assumes that the parsing function for each non-terminal depends only on the input string, and not on any other information accumulated during the parsing process" (Ford 2004)

This algorithm lends itself to incremental parsing because it can reuse the memoization table after an update. The challenge is to determine which entries in the table can be retained. Dubroy and Warth propose a scheme (Dubroy and Warth 2017) in which the maximum examined position is tracked for each parsing function and stored along with the result. If the modification overlaps with any point between the token's start and the maximum examined position, the entry is discarded.

Yedidia and Chong later improved this algorithm (Yedidia and Chong 2021) by switching the table for a binary search tree specialized for intervals, applying the offsets to memoized items lazily instead of eagerly. They memoize at every level of this binary search tree. This accelerates lookups and application of changes while also optimizing the number tree traversals. The optimization lowers the asymptotic complexity of the algorithm to logarithmic runtime based on source file size.

3.2 Layout

Incremental layout recomputation and layout artifact caching is generally most attractive for systems that have to process and display changing data and are expected to respond interactively and immediately. Therefore, such techniques have mainly been researched in and developed for desktop and mobile User Interface (UI) frameworks and web browsers. When compared with the problem and approach in this thesis, the prior art tends to work with a persistent view that is updated instead of Typst's recompilation run, in which a new layout tree is built instead of mutating the old one.

UI frameworks can be categorized as *immediate mode* or *retained mode*. An immediate mode framework gives the application building blocks to draw interactive widgets on the screen. It has to build these widgets from its own state, with its own logic. All processing happens in the main application thread, on every rendered frame, most often by the CPU (Linton, Vlissides, and Calder 1989; Allard and Raffin 2005).

As Nilsson and Reveman 2004 demonstrate, the UI might not be immediately drawn into the screen buffer when using immediate mode. Instead, it is drawn onto a separate layer that can later be superimposed, *composited* (Apple Developer 2018), on top of another pixel buffer, which could be GPU 3D content, or, as in Linton, Vlissides, and Calder 1989, other UI elements. Compositing leads to an early version of layout caching: render rarely changing UI elements on a separate layer that may be composited with the frequently changing UI redrawn for each frame.

The counterpart of immediate mode is retained mode, where an external graphics library will retain the state of the user interface that is then mutably updated by the application (Allard and Raffin 2005). On each call that mutates the widgets in the view, the graphics library can decide whether a rerender should be triggered, when it should happen, and how it is conducted. Crucially, this means that it can treat each

widget separately while keeping this behavior opaque from the application's point of view. Widgets thus only need to be updated if their data changes and can otherwise be retained.

Managing retained mode user interfaces from an application side inspired application architecture patterns such as *Model-View-Controller* which spurred the development of later reactive UI solutions such as AngularJS (Ollila 2021). In a Model-View-Controller (MVC) application, the data (model) and the view are two distinct entities. The view is derived from the data. A controller adds interactivity by providing methods through which the model can be modified (Syromiatnikov and Weyns 2014). Many user interface libraries for desktop applications, such as Apple's UIKit, adopted the pattern, focusing on making the view and model objects reusable (Apple Developer 2018) to, in turn, facilitate library-side widget reuse.

LibreOffice Writer is based on MVC (Vajna 2014). A controller manages document state changes and can be attached to multiple views. The pages in the views are not always painted (especially if they are not in the viewport), and events are dispatched for both repaint and reformat. Some formatting information is cached between state changes (Ama et al. 2020).

Browsers focus on providing fast mutation of the page layout on screen. JavaScript code can, at any time, mutate the displayed document to provide interactivity. The user may scroll, CSS animations might play, and the arrival of deferred resources such as images may cause a change in layout (Vasilijević, Kojić, and Vugdelija 2020).

For its layout process, the browser uses a multi-stage pipeline (MDN contributors 2021; Anderson et al. 2016):

1. It loads and parses the HTML document with its stylesheets and scripts.
2. It creates the Document Object Model (DOM) from the HTML, creates a tree that contains all document nodes, and computes a style tree.
3. From these trees, the position and size of each element are computed in the "Layout" stage.
4. The content and appearance of these elements are determined, and they are drawn onto several layers.
5. These layers are composited together, creating the final frame buffer content for the document.

When the user scrolls, the document merely has to be recomposited (some elements might not have been painted because they were out of view; the browser now needs to paint them) (Lewis 2021).

When the DOM is mutated, either layout, painting, and compositing must all be repeated (reflow and repaint), or, for some changes, the layout step can be omitted (Lewis 2021), reusing the sizings and positions from the previous render.

The omission of reflow is most common when changing CSS properties that do not govern the layout, like `background`, but also for content changes in statically sized elements. Some style changes, such as `transform`, will only require a re-composite in select browser engines, omitting the need to repaint altogether (Lewis and Surma 2018). The CSS property `will-change` allows the web developer to hint the browser to draw an element on a separate layer so that, upon a repaint, no other element must be changed (Atkins 2022).

The Mozilla developer Baron has stated that a governing principle of a browser layout engine shall be "ensuring that the time to respond to small changes is propor-

tionately small" (Baron 2003), at the time mainly as a reaction to layout shifts caused by late-arriving images. This principle became even more critical as the web shifted towards updating the view of web applications with data from APIs, retrieved with JavaScript. In this new design paradigm, developers could minimize DOM changes (Fu et al. 2010). The trend emphasized the need to provide a fast reflow phase not to squander the node reuse potential of this paradigm.

Hence, the reflow stage is split up into two phases: Measuring the new sizes of elements, which only has to be done for the changed elements and all their children (exceptions apply), and repositioning elements, which has to be done for all elements (Panchekha and Harrelson 2021). Between the two, measuring the sizes is more expensive.

The Chrome rendering engine Blink has even further optimized this process by introducing a variety of flags and conditions that leverage the element hierarchy in the DOM and limit reflow calculations, e.g., by using information about the CSS positioning mode (`static`, `absolute`, …). It also retains data about intrinsic sizings of elements to minimize recomputations (Chaffraix et al. 2022).

Zhang et al. have proposed to augment the browser by memoizing the exact layout results for many elements depending on their and the parents' style and content as well as the global state (i.e., size of the browser window). Even after the page in question has been left, such a cache would be persistent. They complement this cache with a derivate DOM tree that groups the children of each node into classes in terms of which CSS selectors applied (Zhang et al. 2010). While the CSS resolution cache can provide benefits, the static layout result cache is very dependent on the environment remaining the same. Reflow optimizations of modern browsers introduced since the paper's publication have likely superseded it.

Performance optimizations across the painting and layout stages enabled a shift in web application architecture: Previously, select parts of a web page might be enhanced with JavaScript and API-based interactions that updated the document, but upon navigation, the DOM was discarded, and a new document was retrieved from a server. Now, a new class of reactive UI libraries such as React and Vue has become popular. Applications built around these libraries only retrieve one document from the server and then continually synchronize the DOM with their underlying data model, which might be populated by an API (Ollila 2021).

These libraries maintain a parallel copy of the DOM (the *shadow DOM*) to which they apply their model changes when they happen. Periodically, the real and shadow DOM is synced. The innovation is that these tree diffing operations can be executed in linear instead of cubic complexity because of heuristics. The React reconciliation algorithm (Alison et al. 2021) performs this diff and merge and uses two assumptions for it:

- Elements that change in kind never share a subtree
- "The developer can hint at which child elements may be stable across different renders with a `key` property." (Alison et al. 2021)

These assumptions mostly hold in practice (Ollila 2021) and enable the library to plug into the browser's performant relayouting systems by providing minimal DOM changes in each step.

This reactive model for element reuse in user interfaces has also taken hold in native applications, as demonstrated by SwiftUI on Apple's platforms and Jetpack Compose on Android. These libraries allow the developer to declaratively build their UI by calling functions in code that mount UI elements and adjust their content to the data

model (Palumbo 2021). These functions are re-executed each time the model changes, such that the "shadow widget tree" is continually rebuilt. Using similar heuristics as React, Jetpack Compose can reuse old widgets with the new state, minimizing the need to create new UI elements or discard their internal state. For this purpose, Jetpack Compose uses a Gap Buffer internally (Richardson 2019). Reusing existing widgets for a shadow widget tree is pivotal because otherwise, the performance cost of rebuilding the tree every time would make the expressive declarative approach to UI impractical.

This approach is taken further by Flutter. Flutter is an cross-platform mobile application framework with a focus on user interfaces (Palumbo 2021). It uses a reactive pattern for its UI and, as opposed to Jetpack Compose, does not defer rendering and layouting its widgets and layout elements to another library. This integration offers opportunities for further integration of layout artifact reuses. Similarly to browsers, Flutter uses layouting, painting, and compositing stages (Barth 2016).

Flutter bases its layout model on sizing constraints (not to be confused with the concept of constraints in this thesis): Elements in a layout element tree tell their children their minimum and maximum size along each dimension. Some element subtrees never change in dimension because they are required to be some exact size. Such constraints make them natural boundaries for reusing parent layouts since they only change if their children change in size. Flutter can easily recognize changed elements in the tree using reactive reconciliation, and only the relevant subtrees are relayouted (Barth 2016).

Flutter's render object tree used for painting also provides incremental capabilities for rerendering. However, the subtrees are not isolated by hand. Instead, the implementor of a widget has to hint which elements are likely to repaint separately from each other such that Flutter places them on different compositing layers (Barth 2016). The combined benefit of the reactive approach as well as relayouting and repainting boundaries helps Flutter achieve near-native 60fps performance on a typical phone (Palumbo 2021).

4 Incremental Recursive Descent Parsing

This section presents an algorithm for incremental recursive descent parsing, one of this thesis' contributions. We will first introduce some terms and notation and motivate the need for such a parser within the context of markup languages and Typst which we chose as the basis for our research and implementation. We then introduce Red-Green parse trees that are the foundation of this approach in Section 4.2. Section 4.3 provides requirements for and a high-level overview of the algorithm. The end of that section drills down into the details of the proposed algorithm. Finally, we informally validate our parser in Section 4.4.

Like in many widely used compilers, the parsing and lexing algorithm of the Typst compiler is implemented as a handwritten recursive descent parser. We investigated whether providing an alternate incremental parser/scanner would be viable for this thesis. Next to speeding up recompilation, an incremental parser also increases the responsiveness of tooling for semantic code completion and refactorings.

The algorithm in this thesis, like most incremental parsing algorithms, operates on modifications on an underlying file instead of being passed the complete, modified source buffer. This way, it can avoid executing a costly diffing algorithm. The change data is most likely already available in the user's text editor, it only needs to be passed to the reparser. For this reason, the algorithm requires an integration between editor and compiler to exchange information about the user's modifications, be it through a plugin or bespoke editor software.

For the following paragraphs, let us introduce some terms and notation. The parser processes a *source file*. The file's original contents are *edited* by a user. The edit consists of a *range* and a *replacement*. The range is a pair of indices in the source file in which the modification occurred, while the replacement is a text string that replaces the original contents of the source file within the range. Outside of the context of an edit, a range characterizes a symbol's position and extent within the source file. The symbol may be a non-terminal. When dealing with formal grammars, we represent non-terminals with capitalized Latin letters and terminal symbols with lowercase letters. The term *clean parse* denotes the parsing of the source file with a regular, non-incremental parser.

4.1 Motivation

Many approaches for generalized incremental parsing are built upon parser generators for LR grammars. An LR parser encodes its overall state through a stack of terminals and non-terminals and a lookahead list (Knuth 1965). Given the previous parse tree and the edit, the old parser state ahead of where the change enters the lookahead can be reconstructed. Consider the following reduction:

$$abcd \rightarrow Ucdef \rightarrow Vd \rightarrow W$$

If the a terminal is replaced by xx, the reduction may look something like this:

$$xxbcd \rightarrow U'cdef \rightarrow Vd \rightarrow W$$

Given the edit's range, the range of each node in the parse tree after xx is shifted one character to the right. Meanwhile, the range of all non-terminals that contain the change (U, V, and W) has grown by one character. The parser can compute the new ranges of all terminals and non-terminals that either fully contain the change or

come before or after it. To then reparse, the parser has to recover its stack up to the position of the first terminal affected by the edit from the parse tree. In our example, the change is at the start, so this step is not needed. After the parser shifts the change onto the stack, b will trigger a reduction to the U' non-terminal. Then, the c terminal triggers a reduction to V. V is a non-terminal that also appeared at that point in the derivation in the original; the clean parse and its range match the expected range given the change. With both the stack and lookahead identical to the initial parse, the parser is now in the same state as it was then with respect to the unchanged rest of the document, so it can reuse the rest of the previous parse tree (only exchanging the V node and its children). With this, the reparse is complete.

By constraining the class of the acceptable languages further, for example, to be Right-to-Left Rightmost Derivation (RL) languages in addition to LR languages, the point at which the parser starts to reuse the old tree can be constrained further (Ghezzi and Mandrioli 1979).

We wanted to investigate further the possibility of using an existing incremental parser technology for Typst with an off-the-shelf generator. Tree-Sitter (Brunsfeld 2018) is a parser generator that allows specifying grammars in JavaScript and then outputs a native parser accessible through C and Rust bindings. Tree-Sitter has a particular focus on AST-based syntax highlighting. Tree-Sitter parsers provide capabilities to facilitate highlighting the code using the AST in ways that a regular expression-based conventional highlighter could not. Under the general procedure explained above, such a parser operates in two distinct stages when reparsing a document. We explain its operating procedure in Section 3.1.

Implementing a Typst parser using Tree-Sitter, however, proved fundamentally impossible. Tree-Sitter can produce parsers for languages that are within the Generalized LR class. This class extends LR(k) languages by non-deterministically running all possible branches of the decision tree when a shift-shift or shift-reduce conflict occurs. LR(k) languages are a subset of context-free languages but GLR parsers can parse every context-free language (Lang 1974). Typst's grammar consists of two parts: The markup syntax and the programming syntax within code blocks and after a `#` in markup mode. Like many other programming languages, the programming syntax is context-free and can be encoded into a Tree-Sitter grammar. The markup syntax, however, is not context-free. Consider a list:

```
- Item 1
  - Item 1.1
    - Item 1.1.1
  - Item 1.2
 - Item 1.3
- Item 2
```

- Item 1
  - Item 1.1
    - Item 1.1.1
  - Item 1.2
  - Item 1.3
- Item 2

In a list with two items a and b, b is a child of a if b is preceded by more whitespace than a. This rule even applies in content blocks; the above list would have the same levels if the statement `#let x = [/* ... */]` wrapped it. Therefore, the parser only considers whitespace within a content block, not the offset from the start of the line in the current source file. Peter Landin coined the term off-side rule for grammars sensitive to whitespace like this in his 1966 article "The next 700 programming languages:" "The southeast quadrant that just contains the phrase's first symbol must contain the entire phrase, except possibly for bracketed subsegments" (Landin

1966). This rule, reformulated a bit, states that every item of an expression must be at least as indented as the start of the expression (and therefore, child expressions must be farther indented than their parents to differentiate between the child and parent expressions). Languages that use the off-side rule cannot be parsed with a GLR parser (Erdweg et al. 2013). Non-incremental parsers typically remedy this by having a stateful lexer convert the whitespace to indentation and dedentation tokens (cf. Python Software Foundation 2021). Tree-Sitter provides an interface for an external lexer written in C that may recognize whitespace and emit tokens that can then be used in grammar definitions, but this would introduce a linear lexing pass over the document, which we sought to prevent by incrementalizing the parsing step.

Instead, we decided to implement a custom incremental parsing method in Typst's manually written recursive descent parser. Typst was implemented with a recursive descent parser to satisfy our goal of providing meaningful diagnostic messages to the users (Mädje n.d.; Brink and Maniero 2010).

4.2 Red-Green Trees

Recursive descent parsers often produce an abstract syntax tree with strongly typed nodes. This is beneficial for later evaluation, however, it makes general traversal of the AST difficult. Furthermore, AST nodes may store information for diagnostics that binds them to their position in a file. These two properties are undesirable for incremental parsing. The tree must be easily traversable to locate the superseded nodes and nodes unaffected by the change should not be invalidated.

Consider the example of a prior implementation of the Typst AST: Its original parser returned an abstract syntax tree (AST) in which each node is strongly typed. For example, the struct definition for an enumeration item within the AST would look like this:

```
/// An item in an enumeration (ordered list): `1. ...`.
pub struct EnumNode {
    /// The source code location.
    pub span: Span,
    /// The number, if any.
    pub number: Option<usize>,
    /// The contents of the list item.
    pub body: Markup,
}
```

The `Markup` type was an enumeration with variants for each of the child types it might have, including expressions. While this architecture leverages the type system to ensure that each AST node's required children are well-formed, it also created some problems. When trying to iterate over all nodes in the tree, custom code was necessary for each of the variants, creating much boilerplate code every time the tree needed to be searched for some node. In incremental parsing, however, effectively iterating, searching, and replacing elements is necessary to use the old AST to either build a new one or update subtrees to match the source code modifications. Additionally, note the `span` field in the struct: It saves the range for this AST node. Suppose changing an AST node before an enumeration item, with the item itself not being affected. Since its position might have shifted due to the edit, we need to update its `span` field and the `span` fields of all the following unchanged nodes.

Instead, we decided to switch over to a new abstraction for the syntax in a source file. Red-Green Trees are a data structure first used in the .NET Core C# compiler (Lippert 2012; Yaakov 2017; Garland et al. 2020). They consist of three levels of abstraction:

- Green nodes only save their length in the source code in bytes, which type they are (for example, an enumeration variant or an integer), and their children.
- Red nodes save a reference to a green node, a `span` of where they occur in the source file, and a reference to the red node of its parent.
- Typed AST nodes wrap a red node and provide convenience methods to cast the children of the contained node into the appropriate typed AST nodes.

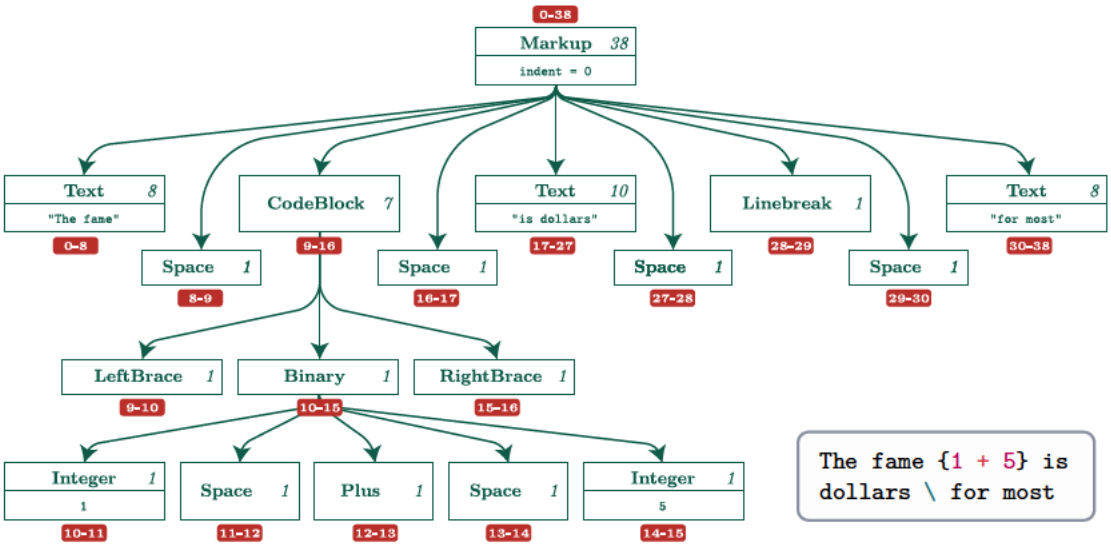


Figure 7: Red and green tree for the code sample in the bottom right. The green tree encodes the type and length of an element and possibly data associated with the type. The red tree transforms the length into absolute ranges in the source file.

The red and green trees are Concrete Syntax Trees (CST) because they contain nodes for all characters in the source file, even whitespace and comments, as shown in Figure 7. Nodes omitted in the Abstract Syntax Tree are also called *trivia*. For every character in the source file, exactly one leaf node in the green tree exists whose range it falls in. The parser constructs green nodes in a bottom-up manner, a good match for recursive descent: each parsing function returns nodes of the corresponding type. After the parsing process, a single green root node contains all other nodes in the tree. A root for a red node tree can be constructed with this node. The red node can then, on-demand, create child red nodes. A red node only constructs the subtrees that a tree consumer requests and the consumer can cast them to their matching typed AST nodes. Accessing the children of a typed AST node cannot fail since the parser guarantees that it will annotate all green nodes whose child subtrees do not match the structure expected for that type. The green tree is intended for the parser only. The red nodes provide quick and convenient iteration over the CST (for consumers like syntax highlighting or automatic refactorings). Other consumers can cast red nodes into typed AST nodes to implement behavior exclusive to some node type.

With this design, the two problems stated above are solved at the cost of two

additional layers of indirection: First, Iteration over the tree is now easily implemented with either green (internally in the parser) or red (externally) nodes. Second, nodes right of an edit need not be updated since red tree construction calculates their position using the length of their predecessors. The parser constructs a CST on a best-effort basis for invalid input, representing an error using a special node kind. Red nodes provide a mechanism for retrieving an error list from their descendants; adding a flag to green nodes indicating whether any descendant is an error accelerates searching for errors.

The idea for incremental parsing now is to find one or multiple neighboring green nodes at the tree's deepest possible level whose source ranges contain the edit. The type of the selected nodes is then mapped to the correct entry point in the parser. However, due to the properties of many grammars, not every insertion that is valid on its own is valid in the context of its neighbors. Even in LR(k) languages, modifying a symbol on the parse stack can lead to different reductions and thus different parse outcomes. Since an incremental parser implementation is correct if and only if it produces the same AST with a source file and an edit that the conventional parser would produce from the edited source file, it must appropriately handle these cases. The design of the incremental parsing algorithm is not motivated by the theoretical properties of grammar and language classes but by the characteristics and requirements common for programming and markup languages. Therefore, we identify the following requirements for incrementally parsing typical programming languages:

1. **The parser must not change inner expressions so that it affects the precedence of unchanged outer expressions without also reparsing them:** Reparsing some parts of composite expressions must not change their precedence within the expression. Consider the expression `x * 5 + 1` which consists of the inner multiplication and the outer addition expression. Using parentheses, the precedence of that expression can be visualized as `(((x) * (5)) + (1))`. If the multiplication sign is exchanged with an assignment operator, the expression would look like this `x = 5 + 1` and the addition must now be the innermost expression `(((x) = ((5) + (1))))`.
2. **The parser must not simply replace nodes that affect their parent's type and children without reparsing the parent and all of its neighbors:** Keywords and operators control the type of their parent nodes and the number and kind of expected children. For example, the input `{let x = 1 {5}}` would produce a let binding, an error because of the lack of a semicolon or newline after a complete expression and a scope with a single, atomic expression in it (the number five). If now the keyword `let` were replaced by an `if`, the node type would change, the expression would contain the block with the number five, and the error would vanish.
3. **The parser must not parse a token that changes the parsing principles without appropriately invalidating their neighborhood:** A common example for this are comments. Inserting a comment can change its leaf siblings and the parents, e.g., by inserting an unmatched `/*` to open a block comment. All parents and their siblings affected by this structural change need to be reparsed in both code and markup mode.

4. **The parser must correctly process inserted and deleted optional children:** Some nodes contain optional last children that may otherwise also appear independently. For example, a call is valid as both `#rect(height: 2cm)` and `#rect(height: 2cm)[markup]`. When exchanging the latter `ContentBlock` node in the call expression, the parser must ensure that it stays the same type or becomes a sibling instead of a child of the expression.

Additionally, we found that parsing Typst poses these additional challenges:

1. **Correct handling of whitespace:** The incremental compilation process must not exchange the whitespace node following a `Newline` for a non-whitespace node so that it won't be confused with an escape node. In markup mode, Typst differentiates between `Text` tokens that contain characters to be typeset and `Space(n)` tokens that encode the whitespace around the text. The n variable of the token encodes the number of linebreaks contained in the space token. The tokenizer will collapse a sequence of multiple spaces and newline characters into a single `Space(n)` token; the parser later converts tokens with $n \geq 2$ into `Parbreak` tokens. The `Newline` token, produced by a single `\` character, forces a new line without starting a new paragraph. It must appear either at the end of a file or followed by whitespace because it would otherwise be an escape sequence.
2. **Correct handling of line starts (I):** The incremental compilation process must not preserve headings, lists, or enumeration items when removing the newline before them or inserting non-whitespace characters between the start of the line and such an item. A heading, list, or enumeration item must always be the first content on their line in the source file. Therefore, there may only be whitespace or comments between them and the start of the line. The sequence `== Approach` at the start of the line would produce a second-level heading "Approach," but if there were a character `a` inserted on the line in front of the two equals signs, it would instead show as "a== Approach," printing the Equals signs.
3. **Correct handling of line starts (II):** Similarly, when a `=`, `-` or `.` (optionally preceded by a number) appears on a source line preceded with some non-whitespace tokens, the incremental compilation process must convert these items into headings, lists, or enumeration items when deleting all non-whitespace tokens in front of them on the line.
4. **Application of the off-side rule:** The parser must correctly nest nodes on lines directly following nodes applying the off-side rule. Headings, lists, and enumeration items apply the off-side rule. If the following source line is further indented than the indentation level that the item itself had, the following line's elements are children of the current node.
5. **Transparency w.r.t. token merging optimizations:** Some edits of the markup would parse differently on their own versus in the context of the previous AST. Consider the markup `twenty changes`, which is parsed into a `Markup` green node with three children: two `Text` nodes with a `Space(0)` node between them. Now consider changing the text to `twenty-two exchanges`. The correct parse has the same number of green nodes as before, with the two text nodes now containing the longer words. Since, however, the change is limited to the location of the space in the unedited source code, a naïve implementation could instead parse the change on its own (`Text(-two)`, `Space(0)`, and `Text(ex)`) and substitute them for the old space token. This reparse would result in four text nodes and not match a clean

parse. Instead, a parser has to adequately replace or amend the old tokens in the neighborhood of the change to make the incremental parse match the clean one. An incremental parser can still provide an behaviorally equivalent parse tree w.r.t. the rest of the compilation pipeline when it fails to fulfill this criterion. However, this failure makes an incremental parser much harder to test with the normal parser.

4.3 Algorithm

We now present an incremental parsing scheme with multi-step selection and self-validating properties to handle these cases appropriately. The algorithm takes an edit and a green tree from the previous compilation and yields an updated green tree that reflects the source file after the edit. To update the green tree, the parser needs to identify and exchange subtrees possibly affected by the change. The tree of parsing function calls during recursive descent mirrors the structure of the green tree, hence only the call subtrees for the selected green nodes have to be re-executed.

A recursive descent parser will return the same thing iff its state, consisting of the input string and any mutable global state is equal. Using this fact, we aim only to execute the segments of the recursive descent call stack that can at all differ between original compilation and compilation with changes inserted. The global state must be easily recoverable from the data in the green node tree. The new nodes resulting from only running the affected recursive descent subtree with the recovered state can then replace the green nodes yielded by the original compilation of that subtree. However, finding a small subtree containing all possible changes for a range of replaced source code is challenging for context-free languages.

The reparsed section of a file is minimal exactly if there is no smaller portion of the file which, when reparsed, would produce the same green tree as a clean parse of the modified file. Our algorithm does not always perform minimal reparses because this would require in-depth knowledge of the grammar being parsed. Instead, for the sake of wide applicability and easy adoption, it errs on the side of caution.

Below, we propose a selection and validation scheme that uses the parser to check whether the old and new parsing subtrees have converged once the parsing of the selected nodes has started.

First, we present a high-level overview of the process in Algorithm 1, starting from the root node of the syntax tree.

---

1. Search for the child node range containing the edit and adjacent nodes that may be affected.
2. If a single node has been selected:
   a. Recurse and try only to reparse its children. If this is successful, return the result and terminate.
   b. If the single selected node is a code or content block, reparse the node with the appropriate function and skip to step 6.
3. If the current parent node is not a markup node, return.
4. Start to reparse the source string from the start of the selected node range.
5. Stop as soon as both of these conditions are true:
   a. The parser processed the source string to at least the end of the selected node range.
   b. The parser yields the first node identical to one in the old syntax tree or

has processed the whole available source string.

6. Check for unbalanced brackets if the edit is not at the end of the source file and fail if there are any.
7. Replace the selected nodes with the parse result.
8. Update the length of all parent green nodes of the change.

Algorithm 1: Incremental Parsing given an edit and the previous green tree.

The following paragraphs describe the algorithm in the context of an implementation. It is implemented as a recursive function that returns the source range corresponding to the exchanged green nodes if it was successful. A function implementing this algorithm receives the complete edited source code, the range of the edit (w.r.t. the original source), the length of the replacement, and the root green node as arguments. The incremental parsing method will then call itself recursively for nodes in the green tree replacing the root. The arguments of these recursive calls track the offset of the current node within the source and whether it is on the path from the root to the tree's rightmost leaf. The latter property allows reparsing when unclosed parentheses and block comments are at the end of the document. The reparsing function will eventually use their mutable references to the current green node to modify the tree in place. If it, and no deeper recursive call, finds eligible children for a node, it will exchange them for the reparsed children.

When called, the reparser will first try to identify the child nodes that need to be reparsed to contain the effects of the change fully. It iterates over the current node's children, producing a source range for each of them. While in this loop, it executes the finite state machine seen in Figure 8. For each non-reflexive transition, it stores the source range of the triggering child node.

The **start** state can be left if the range of the current node contains the edit's range. If the current node fully contains the change, the state machine can either

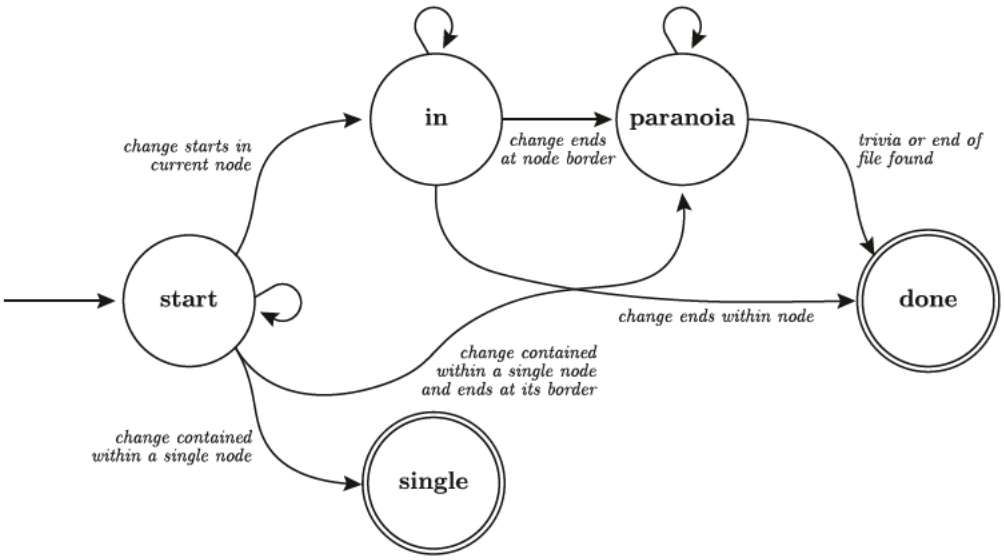


Figure 8: Reparsing Loop State Machine.

go to the **paranoia** state (if the end of the node range and edit range coincide) or the **single** state. If the current node contains the start but not the end of the edit, the FSM transitions to the **in** state. The FSM will remain in the **in** state until the reparser finds a node containing the end of the replacement range (transition to **done**) or coinciding with the end of the replacement range (transition to **paranoia**). The **paranoia** state can be left for the done stage if the loop over all children ends or finds a non-trivia node.

Once the loop completes, we know whether one (end in **single**) or multiple (end in **done**) nodes need to be reparsed and where the source ranges for the first and last node in this *reparsing list* start and end. As described in the algorithm outline above, the parser will parse at least the nodes in this list, continuing until it finds a node for which the old and the new parse tree do not differ anymore. Conceivably, there could be an edit after which the old and new CST share some nodes, even if more changed nodes follow those. The **paranoia** state ensures that the parser will only start to look for convergence until this condition cannot occur anymore by enlarging the reparsing list (paranoidly). For Typst, being paranoid only when the change is not contained fully within some node and adding one additional non-trivia node during paranoia is enough. It should also suffice for most other markup languages. However, implementors will wish to adjust the transitions into and out of this state themselves for their language.

The distinction between single-node and multi-node reparsing lists is important because single-node lists show a fully contained change that may be processed independently from its content. The generic reparsing procedure can be used for reparsing lists of a `Markup` node, while single node reparsing lists offer more freedom. If applicable, the reparsing method will only recurse for single-node reparsing lists and try to reparse some of its children. This recursion may fail, in which case the function attempts an exchange of the node on the current layer. Then, it makes a second effort specific to single-node reparsing lists: If the reparsing list node is a `ContentBlock` or a `CodeBlock`, enter the recursive descent parser with the respective function and try to reparse this node alone, independently of its parent's type. If this reparse has been successful, the resulting node may be inserted according to the procedure below. Otherwise, the general replacement procedure below is used.

If the parent node of the nodes on the reparsing list is `Markup`, multiple nodes may be exchanged at once: First, check if the start of the replacement range coincides with the start of the first node on the reparse list or if the line the start of the change is on is preceeded by a construct with unclear boundries, like a `set`-rule. If so, extend the reparse list by adding either the problematic node or another non-trivia node prior to its start. This is to account for any effects these nodes might have on the parse and must be adjusted when implementing the incremental reparsing algorithm for any other language. In a LR(k) language, the amount of tokens needed to be added here would be k. Then, invoke the parser for children of a content block on a slice of the new source string, starting at the start of the first node on the replacement list. Furthermore, the replacement list end is mapped to a position in the changed string using the length of the replacement. The minimum indentation of the Content Block and the indent at the sliced string's start is passed to the parser.

The parser will start to interpret the source string slice as the children of a content block. Once it has processed enough of the modified source string to reach the mapped end position of the replacement list, it will start to compare the nodes it has yielded so far with the original children of the Content Block. After this position, an early stop

becomes possible: As soon as the parser emits a node identical to a node in the original Content Block's children in mapped position, type, and content, it will stop. This early stop is comparable with the convergence an incremental LR(k) parser achieves when its state and lookahead after a change equal those of the original parse again. If such a convergence is not detected, our parser will continue to process elements until the end of the content block should have been reached.

To solve the thrid problem of tokens with an effect on their neighborhood, the parser then checks if the output contains any non-closed runaway tokens like a string token without a closing quote or an unclosed block comment. Inserting such a token will have consequences for nodes other than the replacement candidates in the green node tree if the rightmost replacement candidate is not on the path from the root green node to the rightmost leaf. Since the search loop and recursion tracked if this is the case for the replacement candidates, unclosed tokens are only allowed if this is the case. Unclosed groups like incomplete statements, blocks, or stray terminators are never allowed because they can indicate that a larger subtree must be considered to yield a valid result and interpret the input in the right way.

If any of these checks fail, the recursive reparsing function call returns without a result to either allow reparsing closer to the root of the green node tree or a clean reparse of the whole source file. Otherwise, the function exchanges the replacement candidates with the replacement list in the child list of the current node. The current node's source length is updated with the source length delta of the candidates and the replacement list. The function finally returns the source range of the replacement list, indicating a result.

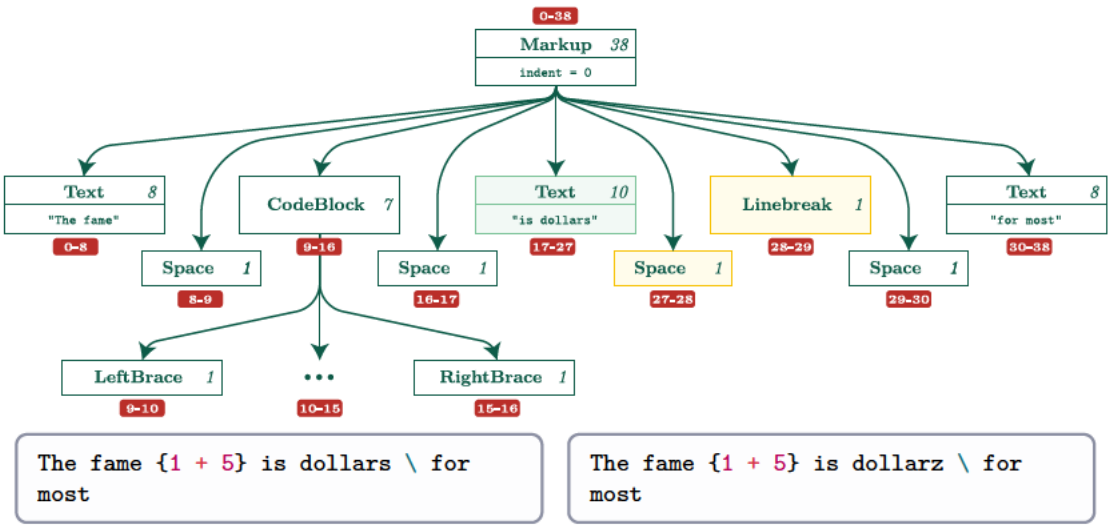


Figure 9: Example for the exchange of the last letter of "dollars." Green color denotes the node the change was in, yellow nodes have been selected paranoidly. The nodes of both colors were reparsed.

Figure 9 shows an example of the reparsing algorithm. When replacing the last letter of "dollars," the reparsing function looks for changes in the children of the top-level markup element. It detects that a text node fully contains the change but requires subsequent non-trivia tokens to be included because the change is at the text token's end. Recursion is not possible because the enlarged reparsing list contains more than one item. Instead, the markup entrypoint of the recursive descent parser is invoked.

All nodes on the reparsing list are processed, as is the following space token. Since this token is identical to that in the original tree, the parser stops here. Then, only the nodes on the reparsing list are exchanged within the green tree.

This paragraph discusses the length of the reparses the algorithm chooses compared with the minimal reparse. The minimal reparse must always contain all nodes directly affected by the change in most grammars. When in markup mode, the reparsing list is, however, extended beyond these nodes in the paranoia state of the reparsing state machine and when conditionally adding non-trivia tokens at the list's start. The amount and size of trivia tokens around the originally selected reparsing list (before entering the paranioa state) determines how many nodes will be added to the reparse list. Sometimes, these extensions of the reparsing list are not needed to produce a correct green tree. Hence, the trivia neighborhood of a node determines how close to minimal the reparses are. For Typst, long reparsing lists can be achieved by editing around interspersed comments and non-comment whitespace, a combination that rarely occurs in real files. For code mode, because whole blocks are reparsed, the reparses are rarely minimal. In Typst, this is not a big problem, because code blocks are intended to be short.

As mentioned above, this algorithm can be adapted for other grammars by changing the behavior of the state machine and the reparsing list enlargement appropriately. Implementations for other grammars that produce wide, flat syntax trees (Markdown, Format Strings, Regular Expressions) will be very similar to the example implementation for Typst above. The algorithm can also be adapted for more traditional programming grammars like Python and Haskell, where Typst's markup token could be equivalent to a block or expression list. There, the state machine needs to be adapted for the effects of keywords etc. by enlarging the enlargement required when in the paranoid state or implementing behaviors more specific to the grammar.

4.4 Testing for Correctness

It is crucial to validate the incremental parser thoroughly because, as explained above, both the left expansion of the replacement list and the properties of the paranoia state in the node search state machine have to be adjusted for the grammar to yield correct results.

We have established a validation scheme in which the incremental parser is tested with the Typst integration test corpus of 114 documents. During unit testing, the documents are parsed clean. Then, one random insertion per 400 source characters (but at least one) is inserted, comparing the incremental reparse with the clean parse of the CST.

After that, another kind of insertion is tested: We insert tokens with a high impact on syntax tree structure (such as closing parentheses, keywords, or spaces) at the boundaries of random CST elements. Both the incremental and clean CST are then compared. This test constitutes a stress test of the incremental parser.

Finally, the test suite is complemented by a set of 71 hardcoded unit tests that cover edge cases of the parsing algorithm in atomic examples. For these unit tests, not only the correctness of the parse result compared with a clean parse but also which nodes were selected for reparsing is checked.

The test suite is executed in Continous Integration after each push onto the main development branch of the Typst repository. The randomized insertions are executed with a fixed seed to make failures reproducible and benefit from the continually growing library of test cases. This test coverage gives us confidence that the incremental

parser is working as intended.

Furthermore, during normal execution, the incremental parser asserts that the total length of its root green node matches the source file length. A failure of this assertion indicates that the incremental parser inserted or deleted nodes from the edit deep within the tree without removing nodes that this edit touched at shallower levels of the CST.

5 Incremental Layouting with Constraint-Based Caches

This section presents an approach to reusing layout artifacts from previous compilations. Layouting is the process of placing some content on pages and in smaller subdivisions on those pages. In the simplest case, layouting arranges inline elements (such as words) in paragraphs and paragraphs and other block level elements as pages. As the Section 2.1 shows, layouting a document can often be computationally expensive and is frequently more challenging than just stacking block and inline elements in their respective direction. Nevertheless, layouters will fundamentally seek to place some content into a pre-allocated space, a *region.* A layout artifact is a fragment of placed content within a region. A complete page layout is made up of several pages, each corresponding to a region and containing multiple layout artifacts. The typesetting software can divide the page region into several smaller sub-regions, as seen throughout this chapter.

Incremental layouting seeks to prevent unaffected elements from repeating layouting after a change. The idea is to provide constraint-based, generalized memoization of layout functions such that if the content does not change and the layout function gets passed the same regions, it will return the same frames from a memoization cache. However, the memoization scheme also attempts to return a cache match if the regions changed, but the layout for the content would remain the same for those new regions. These permissable regions are identified using constraints for the regions that are attached to the cached layout artifacts.

If the inner content of a layout node does not change, it only needs to be relayouted when its containing region has changed. A constraint requiring an exact region match will thus always be valid. However, many other regions will also yield the same layout results. Consider the example for two of the most frequent region changes below, motivating the need to go beyond simple memoization:

- Most pages in typical documents contain multiple paragraphs. The height of a paragraph determines the space left for all other paragraphs on the same page. If a paragraph grows or shrinks by a line, all other paragraph regions on that page are affected.
- In a grid or horizontal stack with cells that resize in accordance to the width of the content contained within, the changes in the width in a cell (for example, by inserting an unusually long word that does not allow hyphenation) affect the amount of space available to the other cells. For example, if any of those cells contained a paragraph, it would need to recalculate its line breaks regarding the new region's width.

These examples illustrate that a change may cause quite a few regions' sizes to change even if their layout node remains unchanged, most commonly when the change is at the top of the document. Naïvely memoized, each constituent layout artifact affected by a region change needs to be recalculated. It can further be observed that, in many cases, the returned frames do not change even with the change in regions. Such cases are easily imaginable in the scenario of the first above example: As long as the space in the current page's region is still enough to fit the whole paragraph (suppose it all fit on one page before), the output frame will not change. Even though, an edit in a paragraph near the document's start triggers a cascade of region changes to the next forced page break. Only if the page's region gets too small to fit all lines the layout function will need to return two boxes, one for each page. When not broken up, the

paragraph only has a minimum requirement for the height of the current page's region to remain unbroken, and it does not need the region's height to remain the same to reuse its old frame.

The same can be argued for the width of a region: Often, when editing tables with numbers in them, the widths of auto-sized cells slightly shift around, not only affecting the edited cell but all other cells in the table as well. However, the paragraphs within these cells only return different frames for the new region widths if the line breaks within change (this assumes the paragraph is not justified). Therefore, the paragraph can reuse its layout result regarding the region width as long as all lines still fit, and no other line breaks would have become possible through increased space.

Because, as the above examples show, cache items are applicable for a wide range of regions and because such equivalent regions are often created when editing a document, the hit rate of a memoized cache would be low even though its items are identical to the recomputed layouts. Our constraint-based, generalized memoization seeks to improve the cache's hit rates because layout artifacts can be reused even if their new region does not match the base region for the original layout. We expect that for a truly interactive editing and recompilation experience in large documents we must leverage constraints, not simple memoization. We also propose a caching and eviction strategy that can be used both for the permissive constraints of Sections 5.1 and 5.2 as well as normal memoization, implemented here as "tight constraints".

We design the constraints such that when editing a document that only consists of text, only the changed paragraph and paragraphs that break or "unbreak" between pages due to the edit need to be layouted again. The cached layouts of all other paragraphs can be moved into the right places, saving a lot of the expensive shaping and bidirectional computations.

5.1 Constraint Implementation

We now examine this scheme further by implementing it in Typst and interfacing with its layout architecture. For this, an understanding of how Typst stores regions, content, and artifacts and how it layouts is necessary.

The Typst compiler outputs a list of frame structs that an exporter can consume and output to a file (e.g., PDF or SVG) or render to the screen (e.g., the Typst web app). Each of the frames in this list represents a page, but they can also be nested, then they become rectangles in the containing frame. Consider this definition of the `Frame` struct:

```
/// A finished layout with elements at fixed positions.
pub struct Frame {
    /// The size of the frame.
    pub size: Size,
    /// The baseline of the frame measured from the top.
    pub baseline: Length,
    /// The elements composing this layout.
    pub children: Vec<(Point, FrameChild)>,
}
```

A frame can contain several children, each positioned at a point within the frame's coordinate system. A `FrameChild` can either be another frame or one of several primitives (glyphs, geometry, raster images, …). The layout phase creates these frames from elements in the layout tree that implement the `BlockLevel` or `InlineLevel` trait. These traits require the implementing struct to provide a layout function to return the appropriate frames. Below is the signature of the `BlockLevel` layout function:

```
fn layout(
  &self,
  ctx: &mut LayoutContext,
  regions: &Regions,
) -> Vec<Frame>
```

The `regions` argument encodes the remaining space in the two-dimensional parent containers (such as pages or linked-together boxes). The function can return multiple frames which appear on (or constitute, in the case of the top-level flow node) multiple pages. Examples of block-level layout nodes include paragraphs, layout grids, and stacks. The `InlineLevel` layout function is similar but works with a one-dimensional width instead of a region as the available size. We will now adapt the layout function to return constrained frames and transparently interact with the layout cache.

To examine such a solution more closely, we need to examine what the `Regions` struct looks like:

```
/// A sequence of regions to layout into.
pub struct Regions {
    /// The remaining size of the current region.
    pub current: Size,
    /// The base size for relative sizing.
    pub base: Size,
    /// An iterator of followup regions.
    pub backlog: std::vec::IntoIter<Size>,
    /// The final region that is repeated
    /// once the backlog is drained.
    pub last: Option<Size>,
    /// Whether nodes should expand to fill the regions
    /// instead of shrinking to fit the content.
    ///
    /// This property is only handled by nodes that have
    /// the ability to control their own size.
    pub expand: Spec<bool>,
}
```

The `Regions` struct is designed to be able to encode one or multiple regions (like pages or chained boxes, such as columns) of possibly varying sizes. The struct can be understood as a self-contained linked list of regions. It always provides a `current` (remaining space) size for the layout function and a `base` size describing its original size, with no other frames placed inside of it. Relative sizing is calculated with the base size. The struct can then be used as an iterator, inserting a new value for `current` and `base` by consuming an element of the `backlog` iterator or repeating the `last` size. The `expand` field tells the layout node whether it should use the minimum space required

or all of the remaining space on the horizontal or vertical axis. Each frame emitted by a layout function corresponds to a single triplet of `current`, `base`, and `expand`, and their region (singular!).

This leads to the definition of constraints in the Typst implementation: In order to specify which regions would lead to a specific frame, one has to specify ranges for possible values of `current` and `base` and a pair of booleans for `expand`. Constraints can contain a minimum and maximum (or exact) target value for `current`'s width and height as well as optional values for the size of `base` along either axis as well as a copy of the `expand` fields of the regions struct that was passed to the layout function that created them. This way, constraints fully describe a single region. The `Constrained<Frame>` struct bundles a frame with its constraints. The layout function will now return the frames along with their constraints. This changes the return value of the `layout` function to `Option<Vec<Constrained<Frame>>>`. A transparent wrapper function that accesses the cache and either retrieves a matching item or stores the fresh layout result is provided such that layout function callers do not have to handle the cache. Section 5.3 explains where the cache resides and how it manages its entries internally.

5.2 Example Constraints

Above, we already discussed some of the constraints that specific layout primitives might set for their frames. This section presents select layout primitives and comprehensively explains the exact constraints they set regarding their layout procedures.

5.2.1 Paragraphs

Paragraphs are the most used layout nodes in many documents. They vertically arrange text, horizontal spacing, and any other layout node at the `InlineLevel` (e.g., images or geometry) into lines. In order to produce a frame, a paragraph first layouts all its child layout nodes to collect their frames. These child layout nodes include, for instance, inline images and geometry. Their text is collected into one string s with the space character inserted for explicit inline spacing and the Unicode object insertion character `U+FFFC` to stand in for any other layout nodes. This string is used to calculate the bidirectionality embedding levels (cf. Section 2.1.2). The text runs, split along the bidirectionality boundaries, are then shaped. The string s is then used to calculate all possible line breaks under the Unicode Line Breaking Algorithm (Chapman 2021).

The paragraph layouter iterates over all available line break opportunities when producing a rugged margin. For each possible line break, it measures how much space a line between this break and the end of the previous line would take up, possibly reshaping if the line cannot be assembled by breaking the previously shaped segments. If the line does not fit into the remaining space of the current region, the previous line break opportunity is used; otherwise, this line is temporarily saved for use should the next break opportunity not produce a fitting line. The layouter moves to the next region if the line does not fit vertically. This is a simple last-fit algorithm.

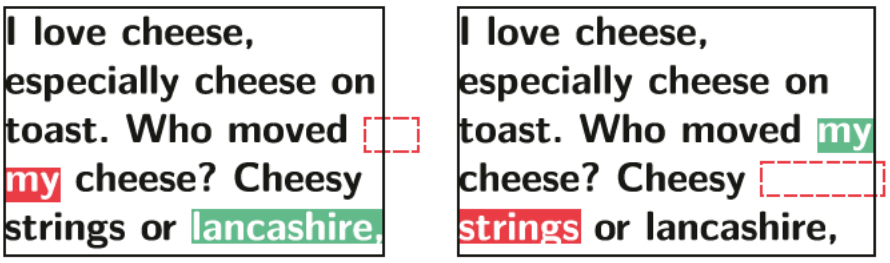


Figure 10: The same paragraph layouted into two differently sized regions. The green word is the last word of the longest line. The red word is the last word on the shortest line that did not fit.

First, we consider a paragraph originally layouted in a region with infinite height. In order to formalize its width constraints, consider the following definitions.

- r^{orig} is the region that is used to create the initial paragraph layout. Its width is $r^{\text{orig}}_w \in \mathbb{R}$.
- n is the total number of lines that paragraph layout produced given the paragraph's contents and r^{orig}.
- m is the number of line break opportunities that UAX#14 produces for the paragraph.
- $f : [1..n] \to [1..m]$ is a function that maps the index of a line to the index of the line break opportunity that ends it.
- $w : [1..n] \times [1..m] \to \mathbb{R}$ is a function that, given the index of a line in the original layout and a break opportunity index, computes the width of the line between the start of the referenced line and the given break opportunity.

Given a paragraph's frame for an original region with height $r^{\text{orig}}_h = \infty$ and width r^{orig}_w, any other regions with infinite height will produce the same frame if their width r_w satisfies the following requirements:

$$\forall\, l \in [1..n] : r_w \geq w(l, f(l))$$

$$\forall\, l \in [1..n-1] : r_w < w(l, f(l)+1)$$

The first condition means that the new region must be wide enough to fit all, especially the widest line. The implementation keeps track of the width of the selected lines to accomplish this and sets the size of the longest one as the minimum width constraint for the corresponding frame. The second condition stipulates that the region must not be wide enough to allow any other, later line breaks to be chosen when compared with the original layout in r^{orig}. The implementation remembers the shortest line that does not fit the region and sets this as the maximum width constraint. Figure 10 illustrates the constraints. Once the paragraph grows wider, "my" is the first word that changes in position. Here, it defines a range of region widths for which the paragraph creates the same breaks together with the last, longest line.

When creating justified paragraphs using the Knuth-Plass (cf. Section 2.1.1) algorithm, the above constraints are irrelevant because the width must always remain the same ($r_w = r^{\text{orig}}_w$) for results to stay reusable.

When working with Regions with finite heights, it can happen that a line cannot fit into a region because it is not high enough. In this case, the above conditions shall be applied separately to the lines of each region during the initial layout process.

A similar constraint is applied for the region's height, with an additional complication: Leading or line spacing must be added between the individual lines. We add the following definitions for this constraint:

- q is the amount of vertical spacing added between the lines of a paragraph.
- k is the total number of regions the original paragraph layout has used.
- $h : [1..n] \to \mathbb{R}$ is a function that, given the index of a line in the original layout, computes the height of the line.
- $r_h \in \mathbb{R}$ for $r \in [0..k]$ is the height of the region with index r.
- $\text{first} : [1..k] \to [1..n]$ is a function that returns the index of the first line set in the region determined by the argument in the original layout.
- $\text{last} : [1..k] \to [1..n]$ is a function that returns the index of the last line set in the region determined by the argument in the original layout.

$$\forall\, r \in [1..k] : r_h \geq -q + \sum_{l=\text{first}(k)}^{\text{last}(k)} (h(l) + q)$$

$$\forall\, r \in [1..k-1] : r_h < -q + \sum_{l=\text{first}(k)}^{\text{last}(k)+1} (h(l) + q)$$

Each of the new regions must thus fit all the lines that the corresponding old region could fit without having space to fit an additional line, if there are any. This condition will thus produce a minimum and maximum constraint for each region that the paragraph spans but does not end in and only a minimum for the last region containing some of the paragraph's lines.

5.2.2 Padding

A pad node adds padding to an element. For this purpose, the pad node contains a `BlockLevel` child and padding as a `Linear`. Padding can be applied separately along each of the four edges of the region. `Linear` sizes in Typst are a combination of an absolute length (the user could specify this in printers' points, centimeters, inches, …) and a relative length that adjusts itself with regards to the base size of the current axis. The function that converts a `Linear` to an absolute length given this base size looks like this:

```rust
impl Linear {
    /// Resolve the relative component to the given `length`.
    pub fn resolve(self, length: Length) -> Length {
        self.rel.resolve(length) + self.abs
    }
}

impl Relative {
    /// Resolve this relative to the given `length`.
    pub fn resolve(self, length: Length) -> Length {
        // We don't want NaNs.
        if length.is_infinite() {
            Length::zero()
        } else {
            self.get() * length
        }
    }
}
```

The padding amount is thus dependent on the width of its regions. If the region sizes vary, the padding amounts do as well if they have non-zero relative components. The pad node shrinks the passed regions for its child and layouts it, and afterward, the resulting frames are enlarged to match the original regions. The amount of padding offsets their position within these larger frames. Whether the pad node can be reused now depends on two factors: The padding must be of the same size everywhere such that the final frame's dimensions match, and the child node must be reusable.

The first condition has consequences for all relative length values that depend on the base size of the region and thus the constraints. In order to satisfy the condition, the pad node must constrain its regions such that a call to `Linear::resolve` always returns the same value. This is the case for any region if the padding contains only zeroed relative components. Otherwise, if either one of the horizontal or vertical border paddings has a relative component, we need to constrain the base size of the regions to the exact base size found for them while layouting. With these constraints, we only need to consider the resolved, absolute length paddings p_{start} and p_{end} for both axes from now.

Axis constraint from padded child	**Axis constraint for pad node regions**
Size of `current` constrained to be equal to passed `current`	Exact size of `current` must equal the `current` size of the original region
Base size constrained to be equal to `base`	Base size must be equal to `base` size of the original region
Size of `current` constrained to be at least s_{min}	Size of `current` must at least be $s_{\text{min}} + p_{\text{start}} + p_{\text{end}}$
Size of `current` constrained to smaller than s_{max}	Size of `current` must be smaller than $s_{\text{max}} + p_{\text{start}} + p_{\text{end}}$

Table 1: How the `PadNode` converts the constraints of its child to its own constraints.

Because a cached frame contains layout elements for all child elements of the corresponding layout node, we have to make sure that the padded child would end up with the same frame. We need its padded region to conform to its returned constraints, and thus, the constraints returned from the child layout process for the padded region have to be transferred to the larger region passed to the pad node.

Table 1 presents a mapping for constraints of the padded child's region to those of the original region passed to the pad node. The transformations in this table are applied independently for constraints on the horizontal and vertical axis.

5.2.3 Grid

The grid node in Typst takes multiple other `BlockLevel` layout nodes and arranges them in a grid of vertical column and horizontal row tracks. It is a particularly complex layout node, so we explain its layouting procedure interleaved with what constraints it sets at each stage.

The cells of the grid fill up with cells first in the direction of the writing system of the user (left-to-right for Latin scripts) and then top-to-bottom. The user can decide how many grid rows and columns there are and what size they have, but there is always at least one column and enough rows such that each child gets its cell. The grid contains gutter tracks between these rows and columns. A grid can break over multiple pages if the vertical space provided to a cell is not enough to fit its content. The definition of the `GridNode` struct looks like this:

```
/// A node that arranges its children in a grid.
pub struct GridNode {
    /// Defines sizing for content rows and columns.
    pub tracks: Spec<Vec<TrackSizing>>,
    /// Defines sizing of gutter rows and columns between content.
    pub gutter: Spec<Vec<TrackSizing>>,
    /// The nodes to be arranged in a grid.
    pub children: Vec<BlockNode>,
}
```

Rows and columns are both encoded as a track here. The `Spec<T>` struct allows to save two different values of type `T`, one for the horizontal and one for the vertical axis. The size of a track is saved as a variant of `TrackSizing`, which can be one of three variants:

- `auto`: Allow the child to grow to fit its content, as far as the other `auto`-sized cells allow it. These children are layouted multiple times, the first time with their regions' `expand` property turned off to determine their smallest "natural" size. In the final grid sizing, they are only shrunk to a smaller size if the total of `auto`-sized cells would prevent the grid from fitting in its region.
- `Linear` values: Absolute and relative lengths. Refer to the explanation of `Linear` in the previous subsection.
- Fractional sizing (e.g., `1fr`): The remaining size of the grid's region is distributed to tracks with fractional sizing. Each track will receive the share of the remaining space corresponding to the ratio between it and the sum of all fractionals. This means that a column with the size `1fr` gets a fourth of the remaining size if the other fractional columns' size (e.g., `2fr` and `1fr`) sums up to `3fr`

$(\frac{\texttt{1fr}}{\texttt{1fr+2fr+1fr}} = \frac{1}{4})$.

Figure 11 illustrates these sizing behaviors. The first ellipse is wider than its column and overflows it while the text is placed into the exactly 2cm wide column. The second, automatically sized column is very wide because its text has been given the opportunity to use the region's width (minus the first columns two centimeters). It will prefer not to do any linebreaks. The reason the third column has any width at all is that the `auto` column tightly envelops its content, providing the leftovers to any fractionally sized columns, in this case, the third.

Even though the grid, like the pad node, has child layout nodes that may return constraints, the grid's constraints are not built on the base of the child constraints but instead built for the grid's region and the track sizings. We chose this approach because the grid's track sizings dictate most of the sizing of child regions, and a combination of constraints for `auto`-sized rows in their multi-stage process would be complicated.

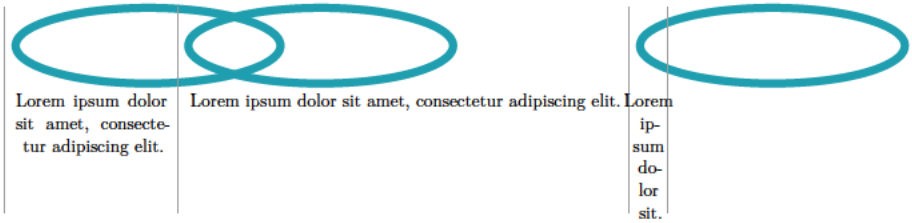


Figure 11: A grid with three columns and two rows. The first column is 2cm wide, the second is `auto` sized and the third is `1fr` wide. All ellipses are 3.5cm wide. The gray lines mark the column widths.

A grid node first determines the width of its columns and then layouts `auto`- and `Linear`ly sized rows one by one. Fractionally sized rows are deferred until the other rows are laid out or the current region is full, at which point they are layouted with the remaining space in the region.

Ahead of the column measurement process, the widths of all columns are initialized to zero. The column measurement process will first iterate over the user column sizings to resolve all `Linear` values to absolute lengths with the `base` size of the grid's first region. If there remains any empty horizontal space in the region, all cells in `auto`-sized columns are layouted to their "smallest" natural size, as described above. For each column, the width of the widest cell becomes the width of the column. Then, one of two things can happen:

- The columns with resolved `auto`-sized tracks fit the width of the grid region: In this case, the remaining space is distributed between the fractionally sized rows as explained above.
- The columns with resolved `auto`-sized tracks are wider than the grid region: In this case, all `auto` rows that take up more than their "fair" share of space are shrunk to their share of the region's width minus all absolutely and fairly `auto`-sized columns already. In fair spacing, all `auto`-sized columns have the same width, and the grid fits its region width.

There are four cases for setting the appropriate width constraint for the current region for this process.

- The user's column sizing only uses `Linear` values such that the grid will always have the same size. The region's width need not be constrained. (If there are non-zero relative components, the region's base size still needs to be constrained)
- The column uses `auto` rows, but the grid takes up less horizontal space than its region even though the automatically sized rows span their "natural" width. The grid will be this wide in each region that is equally wide or wider than the grid. If the region is narrower than the grid, the `auto`-sized rows will shrink, invalidating the stored frames. The minimum width constraint is set to the grid's width.
- The grid exactly fits its region horizontally, which can happen for two reasons:
  - There are `auto`-sized columns, but their "natural" sizes would make the grid wider than its region, thus making a redistribution of their widths necessary, fitting the grid to its region.
  - There are non-zero fractionally sized columns. These columns will always make the grid expand in width to fill its region.

  In both cases, the region's width is constrained to equal the grid's width.
- The width of all absolutely sized columns exceeds the region's width. In this scenario, `auto` and fractionally sized columns are not considered. The grid's output will thus not change until its region's width allows it to at least fit all absolutely sized columns. The maximum width constraint is set to the grid's current width.

Then, the grid measures its rows. Linearly sized rows get layouted with their size resolved over the current region's height. If a linear row does not fit in the remaining height of the grid's current region and more regions are available, it advances the region iterator until it finds a sufficiently large region. During this region iteration, it is made sure that the row can never fit into the skipped regions by constraining their maximum height to the row's height.

The `auto`-sized rows are layouted by first layouting all cells with the column widths and the region's remaining height without the `expand` flag set. This way, the cells take on the shortest possible sizes. The cells, in this process, may nevertheless return multiple frames, indicating that the cell breaks over two grid regions. When a grid contains cells that span multiple regions with no fractionally sized rows, it lays them out in the tallest possible regions, except for the last region. In other words, the grid stretches multi-region rows until the bottom of the regions they do not end on in order to have a uniform height. For all of these expanded regions (every region except the last one) in a multi-region cell scenario, the region height is constrained to equal the original region's height.

Finally, each time a grid region is filled up with rows, the fractionally sized rows of the region are layouted between the other rows. If there is a fractional row, then the height for the region is constrained to equal the region's height. Otherwise, it must at least be the used height such that all of the contained rows fit. At the end of this procedure, the frame for the grid region is assembled from the previously built row frames.

5.2.4 Stack

The stack node is a node that has multiple child nodes that it arranges in a single horizontal or vertical row, optionally inserting spacing between the nodes. The stack struct is, therefore, very straightforward:

```
/// A node that stacks its children.
pub struct StackNode {
    /// The stacking direction.
    pub dir: Dir,
    /// The children to be stacked. They can be other layout nodes
    /// or spacing, possibly fractional.
    pub children: Vec<StackChild>,
}
```

Each of the stack children has its own constraints, which need to be merged into constraints for the stack's frame. Since the stack can arrange its children across either the horizontal or the vertical axis, depending on `dir`, we generalize the terms we use to refer to its axes. The stack axis is the axis along which the stack "stacks," and the unobstructed axis is the other one. For example, the stack axis is horizontal for a Right-To-Left stack, and the unobstructed axis is vertical.

First, since the stack passes its region's base size to its children, it needs to set the `base` constraint for all axes that use relative sizing. If any spacing is relative, it also needs to be set along the stack axis.

There are two cases for either axis:

- Each stack node only specifies minimal or maximal sizes for that axis (they may omit one). In this case, the resulting stack constraints will also specify only minimum and maximum constraints.
- At least one node of the stack specifies an exact sizing constraint for the size of its region. In this case, all child constraints along that axis are discarded, and the stack sets an exact sizing constraint requiring the region's size to match its original region. This case also applies along the stack axis when using any fractional spacing.

The main difference between the two axes is how the stack combines minimum and maximum child constraints.

For the stack axis, the minimum constraints for the separate nodes are summed up, with the frame size substituted for nodes that lack one and the size for spacing children. The calculation of the maximum constraint is more involved: The stack node will allocate as much of the remaining space as demanded to each of its children, unlike the grid node that will redistribute space in `auto` columns to allocate it more fairly. When layouting, every node's region is only affected by the remaining space and thus by the nodes before it. For example, a paragraph will greedily choose the longest possible lines in a horizontal stack, oblivious to any siblings. The frame of that paragraph will tightly fit it such that only a very slim shape could fit next to it in the stack.

The maximum constraint of a node in the stack can be easily translated to a maximum constraint for the whole stack region by adding the stack axis offset of the node. The stack maps maximum constraints for each child that defines them, and

the strictest constraint (the minimal one) is chosen for the resulting set of maximum constraints. Nodes without a maximum constraint can either be assumed to have an infinite maximum size or not be considered in this calculation; both will produce the same behavior.

For the unobstructed axis, we can take a maximum of all child minimum constraints for the global minimum and a minimum of all child maximum constraints for the maximum.

5.3 Cache Structure

Here, we explain how the cache is internally structured and how it interfaces with the layout process. Layout artifacts need to be persistently stored to be reusable. The fastest place to write and read larger amounts of data is the main memory, making it a good candidate for storing the cache. Consequently, any compiler using the constrained layout cache must be long running instead of quitting after each compilation. The cache must support querying for a particular layout node and return the right constrained frames if there was a hit. Optimally, this process is abstracted away from the entity requesting a frame for a layout node. When there is no cache match, the layouter for that node is invoked. There thus needs to be a unified interface with which one can request the frames for a layout node, thus making the cache transparent to all other components. The storage scheme does not differ between caching with tight constraints (memoization) and the permissive constraints described in the previous section.

One of the first steps to take when creating a cache is to make its contents addressable. Since we want to retrieve frames and constraints for an instance of multiple layout nodes, we need to create a unified identifier for all of these layout node instances. This identifier should be equal for two instances of the same type with the same data in their fields but be different for all other layout node instance pairs. We provide hashing functions for each of the layout node types through Rust's `Hash` trait and the Fx Hash function (Matsakis, Stadler, et al. 2020) which was sufficiently performant and exhibited no hash collisions during our tests.

There are several layout node types, but many of them can often be used interchangeably. They are generalized as `PackedNodes`, a container with a heap pointer to the actual layout node and a precomputed value for their hash. This way, the hash of a layout node is only calculated once per compilation. When other nodes need to layout their children, they refer to the general `PackedNode` and use its implementation of the layout trait. The layout trait implementation of a `PackedNode` will try to retrieve the frames for its precomputed hash from the layout cache. It only forwards the layout call to the layout implementation if there was no match of hash or constraints. Before returning a new layout result, the `PackedNode` will store the resulting frames in the cache and trigger the necessary housekeeping routines.

The layout cache itself looks like this:

```rust
/// Caches layouting artifacts.
pub struct LayoutCache {
    /// Maps from node hashes to the resulting frames and regions
    /// in which the frames are valid.
    frames: HashMap<u64, Vec<FramesEntry>>,
    /// In how many compilations this cache has been used.
    age: usize,
    /// What cache eviction policy should be used.
    policy: EvictionPolicy,
    /// The maximum number of entries this cache should have.
    /// Can be exceeded if there are more must-keep entries.
    max_size: usize,
}

/// Cached frames from past layouting.
pub struct FramesEntry {
    /// The cached frames for a node.
    frames: Vec<Constrained<Rc<Frame>>>,
    /// How nested the frame was in the context
    /// it was originally appearing in.
    level: usize,
    /* additional omitted fields */
}
```

The central part of the cache is a hash map from layout node hashes to possibly multiple vectors of constrained frames (a cache entry). A layout node can produce multiple frames in a single layout process, for example, when it breaks across pages. Hence, the `FrameEntry` contains a vector. The regions of the query must all match an entries frame constraint. Nevertheless, since the cache is designed to store multiple cache entries per layout node instance, another vector of frame entries is needed.

Multiple entries per layout node instance are allowed because a single layout node can produce different results for different regions. Nodes like the grid node will layout some of their children multiple times with different regions in their layout process. The design goal of the cache is to prevent the relayouting of a node altogether if neither it nor its usage sites change. Hence, a grid cell must have multiple cache entries to, e.g., prevent a relayout when another child of fixed size in the containing grid changes. Keeping multiple entries for a layout node also has the effect that the cache can retain entries from past iterations of the document from the user's editing process, accelerating operations like undo. The cache can also accommodate a version of a paragraph that remains unbroken between two pages and a broken paragraph with two frames. If then, for example, the content ahead of the breaking paragraph grows high enough, the paragraph gets pushed onto the next page and starts reusing the entry with the single, unbroken frame.

Figure 13 in the next section provides a visual overview over the cache structure.

5.4 Cache Eviction

Regardless of whether tight constraints (memoization) or permissive constraints as outlined in Section 5.2 are used, the cache fills up continually with both entries for new hashes and new entries for old frames when editing a document. To stop the cache from growing ever larger and exceeding the user's available memory, it, at some point, has to start to remove unused elements.

A general caching problem is to remove enough elements for the cache to not exceed a maximum size (all elements are assumed to have the same size). The selection of these items should be such that there are as few as possible cache misses in the future. To achieve this, it is best to remove the items which will be required at the latest moment when compared with the other cache entries. This clairvoyant algorithm is of course impossible to implement in systems without perfect knowledge of the future, such as Typst.

In practical situation, the Least Frequently Used (LFU) and Least Recently Used (LRU) policies perform well enough and are easy to implement (Silberschatz, Galvin, and Gagne 2008). For our cache, we try to improve on the well-known cache eviction algorithms by taking the typical user behavior of someone editing markup into account.

We optimize our cache eviction policies for a good user experience. Especially for performance perception, the user experience is shaped by whether an application meets or exceeds the user's expectations (Bouch, Kuchinsky, and Bhatti 2000). We assume that the user has the following expectations when editing and designing a document:

- They expect to be able to quickly undo recent edits.
- They expect "small mutations" of the last few states of the document to compile quickly.
- They allow for longer compilation cycles when adding, removing, or mutating a big amount of content in a single edit.
- They allow for longer compilation cycles when adding "complex" content, such as large raster images or the output of functions invoking a lot of Typst code (e.g. graph drawing) but expect subsequent compilations to be "as quick as it normally is."

From observations widely made about caches in general and the assumed performance expectations of our users, we can determine what properties of cache entries may be a good predictor about their future use:

- Items that have been accessed frequently in the past will likely be accessed frequently in the future.
- Items that have been accessed in the last few recompilation cycles are more likely to be reaccessed in the near future (temporal locality). This especially means that items that have been accessed or created in the last compilation cycle will very likely be accessed again.
- Items that have been accessed multiple times in a single compilation are used or layouted multiple times in the document and are thus likely to be accessed again.

To consider all these factors, we base a cache eviction algorithm on the LFU scheme and enhance it with data about the access patterns in the most recent compilations.

If the access patterns have some specific form after a compilation, the corresponding cache entry must not be evicted, even if it is less frequently used than other items. By choosing the pattern properties that lead to the eviciton block carefully, we can retain the elements needed to keep the user's performance expectations.

Recent accesses are stored in a short array (e.g. Typst uses arrays with five elements). The array stores the hits for each of the last four compilation in it first four entries. The last entries stores the hit count of all prior compilations and is called the collector. Insertion of an element into the cache does not count as a hit. Along with this array, the number of compilation cycles this element already exists for is stored.

For these *entry pattern* arrays, we now define some properties:

- **Only Zeroes:** The entry pattern consists entirely of zeroes.
- **Multi-Use:** The entry has been used more than once in a cycle, as witnessed by a greater-than-one entry in the non-collector fields or a value in the collector field indicating that the entry was used more than there were compilation cycles.
- **Active:** The first element of the entry pattern array is not zero, showing that the entry was used in the most recent compilation.
- **Sparse:** The entry pattern contains zeroes to the right of non-zero entries.
- **Abandoned:** Two or more zeroes are at the front of the entry pattern array, meaning it has not been used in recent compilations.

In addition, an entry can have the two properties **mature** and **top level.** A mature entry exists for as least as many compilation cycles as the pattern array is long. A top level entry is the root entry of the layout tree. Figure 12 shows example entry pattern arrays with their properties.

Figure 12: Sample entry patterns annotated with their properties. The most recent accesses are found on the left side.

We now designate some conditions with these *entry pattern properties* under which an entry cannot be removed from the cache.

First, we want to enable the user to undo all recent changes quickly. Because of this, we want to keep an undo stack of top-level frame entries to return to for at least the length of the pattern array. We accomplish this behavior by requiring to keep top-level entries that are not mature. To improve performance for recent changes, we require that entries with non-mature all-zeroes patterns be kept. Keeping such entries will prevent the eviction of content the user types (layout and cache entry creation) and then wraps in some function. While typing the function, the content may disappear, but once the user finishes typing, the content could reappear and should be retrieved from the cache. Figure 13 illustrates how the cache contains at least four of the top-level stack elements, one for each of the most recent edits. This way, the document states they correspond to can be quickly recovered from the cache without relayouting. Since the top-level element is mostly layouted into the same region (page size) and

since its contents change with each edit, each of the stack entries contains only one version. Because the top-level elements that are not used for undo operations are not hit very often, the stack will be evicted soon after it becomes mature. Contrast this against the paragraphs in the cache, some already fairly long-lived with one or more cached results.

The following conditions introduce some components of the LRU scheme: We want to emphasize keeping items that recent compilations have used multiple times because they are likely to be reused, possibly multiple times (e.g., some new grid content or page header). Caching these items has a high impact on compilation times and should be prioritized, even if they were stale in the past. The corresponding condition is to keep entries that are multi-use and not abandoned. Building on this, items that were active, i.e. hit in the most recent compilation should not be evicted.

Finally, we want to keep entries with sparse patterns. They are often regularly but intermittently used, and this usage pattern can be penalized by the LFU scheme. We want to counteract this for recent sparse accesses by forbidding eviction of entries with sparse patterns.

After each compilation, the compiler must call a cleanup routine. There, the cache refreshs the number of cycles each entry has been alive and their properties. The cache will evict all removable elements according to LFU until either the cache has the maximum number of permitted entries or there are no other removable entries.

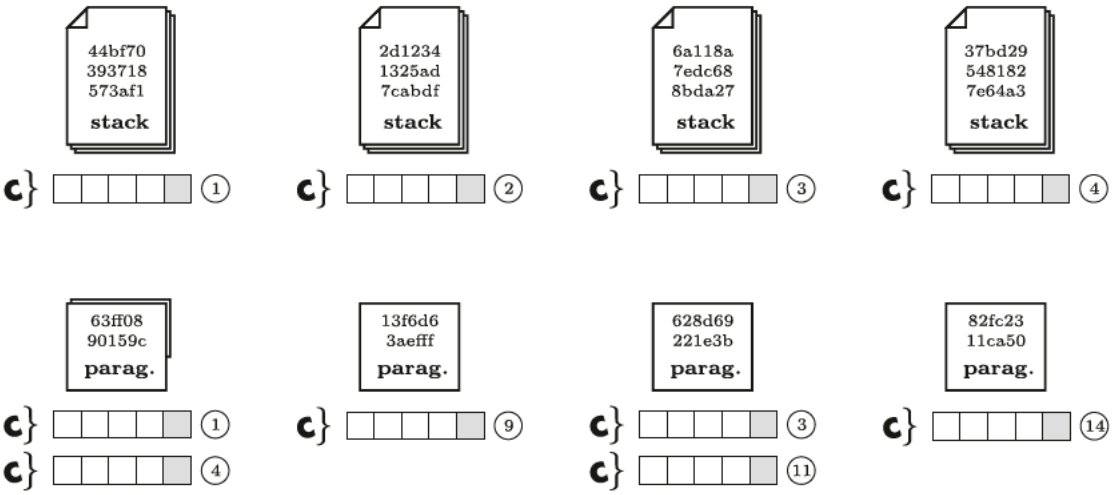


Figure 13: Overview of the cache items. A hash acts as the lookup key for cached layout artifacts. Each artifact can store one or multiple layout results with the corresponding constraints (visualized as the 'c'). For each of these cache entries, a pattern array and the time it has been stored in the cache is saved.

The amount of memory that is freed by evicting a frame can vary greatly. Not only are there frames like images that might store much data, but frames may also contain other frames. Because the elements of a frame are stored on the heap behind reference-counted smart pointers, they are destroyed when the last frame in the cache referencing them is dropped. Because frames can be nested and the both the child frame separately and the child with its parent is cached, some elements can be quite long-lived. Large amounts of data are often dropped when a top-level page frame is evicted from memory. Because this only happens after Typst provided the user with

its output and Typst usually does not run on the UI thread, this does not have a large performance impact.

We chose a cache size of 2,000 frames for our evaluations because of their good performance seen in Section 6.3. At the same time, this amount of caching results in a reasonable amount of memory use, consuming under 750MB even for large documents.

5.5 Testing for Correctness

The constraint-based layout cache works correctly if the constraints of each frame entry are restrictive enough such that no region that would result in another frame for the associated layout node matches the constraints. This is trivially true for tight constraints, since they only match the frame's original region. However, this property has to be checked for more permissive constraints. Furthermore, for the cache to deliver any positive results, as many regions as possible should match an entry's constraint if using them to layout the corresponding layout node would result in the cache entry's frames. In particular, the original frames used for layouting must match the constraints. If this were not the case, the layout cache could be less effective than the simple memoization of the layout functions.

We implemented an automated testing procedure that validates the above conditions for Typst's implementation of the layout cache. Multiple test runs are executed on each document in the Typst test corpus consisting of 114 documents. In each of these runs, only the entries for one level of the layout tree are retained. After the recompilation, the test runner checks whether all retained cache entries have been hit. It also checks whether the recompilation result is identical to the original clean layouting result.

This procedure can catch two kinds of errors:

- An entry has not been hit: The corresponding node was not layouted or that the regions it was layouted with did not match the constraints of the cached entry. This error indicates that the node layouter emitted constraints that would not fit the region provided initially.
- A mismatch between the recompiled layout and the original layout: There was an entry in the cache that was used for a layout node even when the regions would produce a different frame layout. Here, some constraints are too permissive. These error conditions most frequently occur in tests containing grids or other containers that have to layout the same node multiple times with various regions.

The test procedure has proven effective for eliminating constraint setting errors from the various layouters.

6 Evaluation

This section contains results for the quantitative performance evaluation of both the layout cache and the incremental parser regarding the speedup they generate for Typst (Section 6.2) as well as a whole-system comparison to all common LaTeX compilers (Section 6.3).

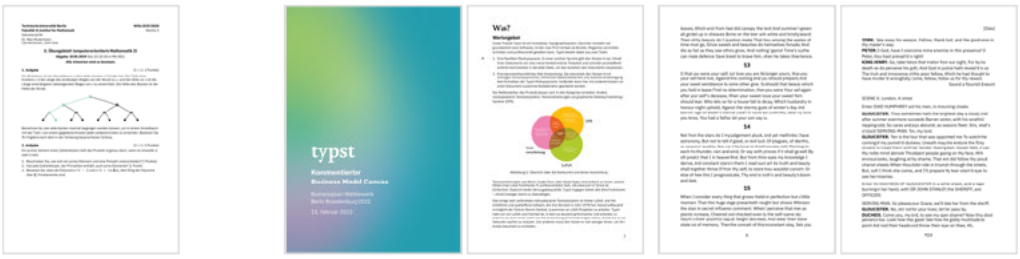
6.1 Evaluated Documents

For our benchmarks, we define three reference documents and three reference transactions. Transactions can consist of one or more actions, each action being a modification of the source file of the Typst document. Each document will be tested with each of the transactions. The speedup average for all actions in the transaction will be reported in relation to the clean compile time of the document before the first step of the transaction.

The benchmark suite contains the following documents, implemented in Typst and two flavors of LaTeX (for pdfTeX and XeTeX / LuaTeX, respectively):

1. **Activity Sheet:** A single-page worksheet as often found in schools and universities. It is a lightly formatted A4 page with a single PNG raster graphic and a list.
2. **Brochure:** A 20-page document that mainly consists of text, embeds multiple high-resolution raster graphics, as well as SVGs, and makes heavy use of `set` and `show` rules.
3. **Tome:** A selection of Shakespeare's work, on 1707 pages in ISO C6 size. There are no images used. Headings, alignment, emboldening, and manual page breaks are the only formatting.

We chose these documents as a representative cross-section of documents set in Typst in content, formatting structure, and length. Figure 14 shows screenshots of them.



Activity Sheet **Brochure** **Tome**

Figure 14: Sample pages from the benchmark documents.

To test the incremental properties of Typst, we the benchmark suite contains the following three transactions:

1. **Typing:** During this transaction, a paragraph is inserted character by character, recompiling after each of them. The transaction consists of 775 separate actions, and the paragraph will always be inserted at the document's start.
2. **Grid:** This transaction inserts a grid with various rectangles, graphical, and programmatical elements in a single action. The grid is complex from a layout perspective, and the child layout nodes have to be layouted multiple times to determine their width in the final frame. The grid will always be inserted between two paragraphs in the middle of the document. Figure 15 shows the inserted grid.

3. **Deletion:** This transaction deletes about 300 characters paragraph from the end of the main body of text (not considering appendices in case of the brochure) and reinserts them all in the last step. Like in the typing transaction, the characters are deleted one by one, but the final reinsertion is a single action as if the whole paragraph was pasted again.

As with the documents, the transactions should represent the variety of actions a typical user could take in a source file: They range from simple keystrokes entering body copy to pasting a section of complex code in a single operation.

The source code inserted in the first two transactions can be found in the appendix; the source files for the documents are available on https://mha.ug/master-files/.

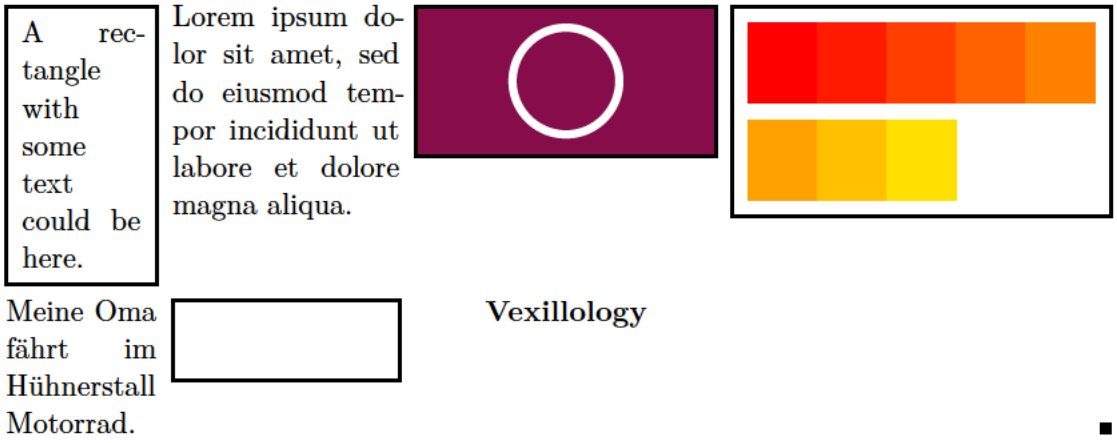


Figure 15: The result of the code inserted in the grid transaction.

6.2 Performance Measurements

We want to provide meaningful and stable numbers showing performance differences at every stage. While the meaningfulness stems from the cross-section of documents and edits evaluated, the stability shall be provided through a well-designed test methodology. In this evaluation, both the total runtime of the program and the runtime of separate stages have to be measured. However, later stages of the program cannot be executed without the prior ones' because they require the previous stage's output.

6.2.1 Methodology

We perform our tests with Cachegrind, a virtual CPU-based profiler from the Valgrind software project to minimize benchmarking noise. Cachegrind will instrument a binary at runtime and run it on a simulated processor with user-specified Level 1 (L1) and Level 3 (L3) cache sizes. It logs each CPU cache access and whether it was a hit. Based on how many cycles a processor has to idle for an L1 and L3 cache miss and the elapsed instruction cycles, Cachegrind can estimate the number of cycles required to run the program. Since the simulated processor eliminates many noise sources of timed measurements with real hardware, the results are much more stable. SQLite has based its benchmarking on Cachegrind, achieving a speedup of 50% over roughly six years by accumulating micro-optimizations (SQLite 2019).

Modern CPUs are pipelined, use multiple caches, have dynamically adjusted clock speeds, and perform branch prediction, all of which, together with the scheduling of

the operating system, introduce noise into wall clock or CPU time-based measurements (Turner-Trauring 2020; Rohou, Swamy, and Seznec 2015). Hence, we decided against those benchmarks for our primary evaluation because their noise level between runs is often too high to measure any changes in performance accurately. The result's noise is even larger when preparatory work has to be excluded from the benchmark to measure the runtime of each stage separately.

In order to have repeatable measurements for the same binary and input under Cachegrind, we wanted to remove all remaining sources of noise: Loading source and image files from mass storage necessitates Syscalls so that the operating system issues a file handle. The busyness of the operating system and the disk, amongst other factors, can influence how many cycles this adds to the measurement. Since none of the contributions of this thesis are I/O bound, we chose to exclude this source of noise by embedding all fonts, images, and source files required to run the benchmark inside of the benchmarked binary. Once the binary starts, they are instantly accessible at a static address, eliminating the need for syscalls here.

The second source of measurement noise within our application was the `HashMap`. Rust initializes the hashing algorithm of its hash maps with a random seed for each instance to prevent Denial of Service attacks (Edge 2012). The Rust random number generator, in turn, is seeded by the operating system's source of randomness (e.g. `/dev/urandom`). The `HashMap`'s randomized layout will change when L1 and L3 misses happen and the order in which the map yields its elements. We compiled a custom version of the Rust standard library which seeds all hash map hashers with the same static seeds.

Finally, Address Space Layout Randomization (ASLR) is a feature of modern operating systems that randomizes the base, heap, stack, and library positions of a binary to hamper the creation of reusable software exploits. It, however, can change what data and instructions are close to each other in memory when the application binary is loaded and hence the hit/miss rate of the CPU caches. We disable ASLR to get more repeatable measurements.

This measurement process can report the same measurements between two Cachegrind runs of the same executable with the same data.

Alas, this measurement procedure is not without faults. First of all, the simulated CPU does not fully emulate the behavior of a real, modern CPU. For once, the estimated cycle count does not correspond to the measured performance on a single, real machine, but it does correlate well, as corroborated by SQLite 2019. Additionally, Cachegrind, by default, performs no branch prediction. A branch prediction switch is available, but according to the manual, it will simulate a severely out-of-date Pentium 4 (around 2004) processor. The branch predictor increases the run time of the benchmarks significantly without providing meaningful insight into how a processor with a more sophisticated branch prediction algorithm might perform on the code. Therefore, the contribution's friendliness to modern branch predictions is hard to evaluate.

We ran the benchmarks on an x86-64 Intel machine with a GNU/Linux binary. However, much of the envisioned use of Typst will happen through its WebAssembly binary. Any differences in performance the architecture may cause are invisible to these benchmarks.

Finally, the measurement of a single stage is not without error. If the estimated cycles $d(B)$ for stage B are deduced by measuring $d(A + B)$ and $d(A)$, then we assume that we can use these two measurements: $d(B) = d(A + B) - d(A)$. In reality, we made the experience that, based on input length, there is an error between 0.0001%

and 0.009% in these calculations. Considering the performance data we gathered, this error is negligible.

6.2.2 Results for Parsing

Table 2 shows the speedups achieved by incremental parsing instead of conventional, clean parsing without a prior CST. All document-transaction pairs show significant speedups, ranging from 1.88x to 19.73x improvements. However, the speedup varies much between transactions, owing to the properties of the incremental parsing algorithm.

	Activity Sheet	Brochure	Tome
Typing at the start	8.44x	4.02x	3.72x
Grid in the middle	1.88x	3.88x	7.94x
Deletion at the end	12.90x	19.73x	13.12x

Table 2: Speedup for parsing for combinations of sample documents and modifications.

First, the speedup in the typing scenario decreases the longer the document gets with the activity sheet seeing the largest and the tome the smallest improvement. We explain this observation with the storage of green nodes in the Typst implementation: Each green node stores its children as a Rust vector, meaning that they occur in order on the heap starting from a specific pointer address. If, e.g., the number of nodes at the start of the vector increases, all other elements have to be moved to the right to make room in "the right spot" for the new element. Similarly, all elements in the vector have to be shifted left if an element at the front gets deleted. This shift creates a linear penalty regarding the number of child nodes for inserting or removing new nodes at or near the start. A Typst AST is usually very flat, with the root markup node having a disproportionate amount of children. Hence, the necessity to backshift imposes a lower bound on the speedup where the backshifting is a significant part of the runtime of reparsing. This lower bound gets less severe the further back a change appears in the children of a node.

The grid example is syntactically complex and shows some of the smallest speedups in the experiment. We expected this result because, for this example, the parser must use more of its code paths than just those to insert mere text nodes. All documents consist mostly of text, which is relatively easy to parse. The call stack of the recursive parser is shallow and few nodes are produced. The grid transaction is a contrast. It consists of expressions and multiple nested code and content blocks, producing many small nodes and a deep green tree and parse call stack, making it more expensive to parse. The activity sheet shows an especially small speedup because the code inserted in the grid transaction is roughly as long as half of its source. Generally, this is a scenario where a large chunk of previously unseen content is inserted instead of just single-letter adjustments, explaining why there is more reparsing work to do per cycle and hence less of a speedup. As expected, the speedup scales with the document's length here.

Finally, the deletion transaction shows similar speedups for all documents. It incurs comparable workloads in the respective documents, and effects like the vector backshift penalty do not apply because of the position of the changes in the document.

6.2.3 Results for Layouting

Table 3 shows the speedups for the layout cache with constraints. All document-transaction pairs show significant speedups compared with the original layout with an empty layout cache. The data indicate that larger documents with longer paragraphs benefit more from the layout cache than smaller documents. The activity sheet has the smallest performance gain because the content inserted into it made up a larger proportion of the document than for the other documents. We must note that each letter inserted or removed from a paragraph will change that paragraph's hash and trigger a relayout. Therefore, in the insertion and deletion transactions, at least the whole affected paragraph has to be relayouted.

	Activity Sheet	Brochure	Tome
Typing at the start	11.24x	83.96x	61.86x
Grid in the middle	4.80x	32.71x	47.49x
Deletion at the end	17.35x	82.94x	62.68x

Table 3: Speedup for layouting for combinations of sample documents and modifications.

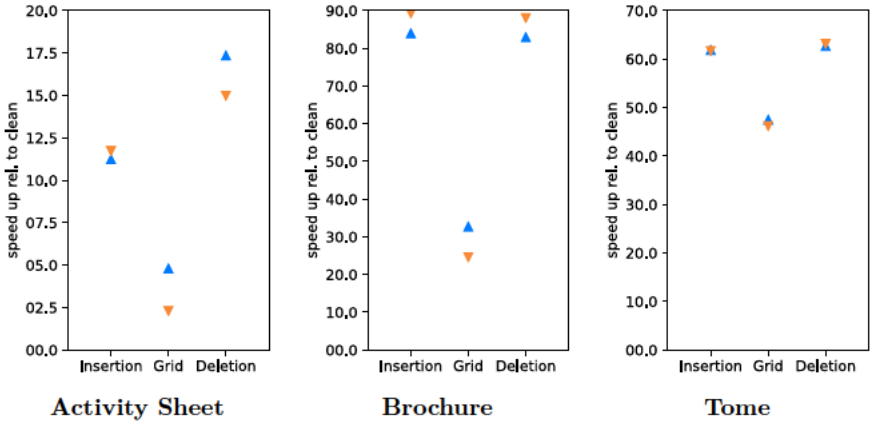


Figure 16: Speedups for the layout cache with permissive constraints (▲) vs. tight constraints (▼). Higher is better.

According to this logic, the speedup should be the largest for the tome document. The table shows this not to be the case, however. An explanation for this is the maximum size of the layout cache: It is limited to 2.000 entries, but the tome contains more than 4.200 paragraphs. Because of the access-pattern-based eviction policy, the entry threshold will be exceeded but not enough to store all paragraph nodes. Typically, storing nodes containing multiple paragraphs higher up the hierarchy would remedy this limitation. For the tome, however, the layout tree is very flat and has few nodes (e.g. `AlignNodes`) containing more than one paragraph.

Figure 16 offers a comparison between speedups measured if using the constraints described in Section 5.2 ("permissive") as opposed to performing function memoization, realized here by issuing constraints that require the region to match precisely ("tight"). Both approaches use the cache structure and eviction mechanisms presented in Sections 5.3 and 5.4. In five of the nine measurements, the speedup with the permissive constraints was more significant than with the tight constraints. The difference

in speedup is most pronounced in the grid scenario. Here, the layout cache can be of use even in the first layouting process of the grid because the grid layouts its children multiple times with slightly differing regions. Some of these relayouts can be skipped when the constraints of previous layout processes still match.

The differences are the least pronounced in the tome document because the need to relayout entire unchanged paragraphs not stored in the at-capacity cache is a significant part of the work necessary for both approaches. It is also notable that tight constraints sometimes narrowly outperform permissive constraints. Some work is needed to compute the permissive constraints they have to "earn back" by making the cache hotter. However, these differences may also be due to different compiler optimizations for the two binaries with different constraint strategies.

This illustrates the strengths and weaknesses of permissive constraints. Nodes with complex layout behavior, slight changes in the dimensions of a container, and changes causing widespread new paragraph breaks can profit from a hotter cache. In other scenarios, simple memoization is enough to make previous results reusable, and the computation of permissive constraints slightly slows the compilation down.

6.2.4 Combined Results

The total speedups gained from the incremental techniques presented in this thesis can be seen in Table 4. It contains the speedup factors calculated by dividing the sum of the estimated CPU cycles of parsing $d(P_i)$ and layouting $d(L_i)$ in the initial clean compilation by the sum of average estimated cycles for $d(P_a)$ parsing and layouting $d(L_a)$ each action.

$$\frac{d(P_i) + d(L_i)}{d(P_a) + d(L_a)}$$

This table shows that the layouting performance increase is the main contributor to the total speedup. The parsing only starts to have an observable effect for larger documents, where it helps to maintain the considerable performance increase. In general, the reported performance increases lie between 4.56x and 80.00x.

	Activity Sheet	Brochure	Tome
Typing at the start	11.17x	67.94x	48.99x
Grid in the middle	4.56x	30.05x	43.82x
Deletion at the end	17.21x	80.00x	58.93x

Table 4: Weighted average of speedups for incremental parsing and layouting.

Due to the nature of Cachegrind, we did not, so far, provide any durations that Typst needs to compile documents. However, the speedup factors allow us to calculate what "time budget" is available for initial compilations in order to hit the well-known 30 and 60fps performance targets. With these numbers, Typst can take between 0.5 and 6 seconds for the initial compilation of a document to maintain 60 frames per second while editing, not considering rendering and raster graphics and allowing the typical time for the evaluation phase. To achieve the more modest performance of 30fps, these budgets can be doubled. Hence, a considerable performance runway is available. Because the rendering of the frames is not measured in this benchmark and

	Activity Sheet	Brochure	Tome
Typing at the start	34.40x	375.47x	38.10x
Grid in the middle	16.10x	193.05x	32.93x
Deletion at the end	65.86x	423.95x	44.04x

Table 5: Total speedups for incremental compilation.

can take up considerable time, these budgets should not be fully depleted in real-world applications.

Table 5 shows the real-world incremental speedup of Typst when parsing, evaluation and layouting are executed. The exporter is not included here. This table is the first to include the evaluation, and thus shows additional speed improvements for documents with raster graphics. Both the activity sheet and brochure documents report higher performance increases than achieved through either incremental parsing or the layout cache, brochure extremely so. Speedups in the evaluation stage that are not a part of this thesis' contributions explain these performance gains: Typst will decode each raster image it loads in the evaluation phase and store the decoded bitmap buffer for future compilations using the same graphic.

Both documents make use of raster graphics. The decoding process for PNGs and, more so, JPEGs takes up a significant amount of time in the initial evaluation, explaining the considerable speedup. The skew introduced by this optimization makes this table unsuitable for ascertaining the total performance gain provided by this thesis' contributions, hence the reader should mainly refer to Table 4 for this. All tables in this subsection except Table 5 exclude this effect.

6.3 Comparison with LaTeX

In this section, we want to complement the previous evaluation with comparisons with various LaTeX engines. I/O often determines real-world LaTeX performance due to the various loaded formats, document classes, and packages. It is challenging to instrument TeX executables like Typst in the benchmarks above. Therefore, we conduct the benchmarks in this section using wall-clock time.

We compare the widely-used pdfTeX engine and the modern XeTeX and LuaTeX engines with Typst (both incremental and non-incremental / clean). For this purpose, we use the LaTeX versions of the *Work Sheet*, *Brochure*, and *Tome* documents. Work Sheet and Tome use the default Computer Modern fonts, but the brochure uses the same OpenType fonts as the Typst version when compiled with XeTeX or LuaTeX. Work Sheet loads six packages in the pdfTeX version and four otherwise, Brochure loads 16 / 13, and Tome 5 / 3. In the XeTeX and LuaTeX versions of the benchmark documents, the appropriate packages (`polyglossia` instead of `babel`) are loaded, and `fontenc` and `inputenc` are omitted.

The benchmarks use shortened versions of the *Typing* and *Deletion* transactions mentioned above.

We used the following procedure for measuring an action with a TeX engine:

1. Write the source file with the current change in a temporary directory
2. Compile the source file to PDF with the engine under test
3. If the engine requests it, rerun the compiler to settle labels
4. Retain all output files of the engine in the temporary directory (`.aux`, `.pdf`, `.log`, …), allowing TeX to skip label-induced recompilations after the initial compile sequence

We complete these steps for all transactions' actions and measure the time for completing the whole transaction. The Typst measurements are instead taken differently: Because the Typst binary is designed to be long-running and retain its incremental compilation artifacts in memory, the benchmark binary links against Typst. For the incremental compilation, the benchmark runner passes the source string or the modifications to it within the same process. The first compilation is a clean one, all subsequent compilations that are triggered by edits are incremental. For the clean compilation benchmark, the runner discards Typst's context, including the green tree, the layout cache, and the raster image cache between each compilation and passes a new, edited source file at every step. In both evaluation modes, Typst, like LaTeX, still has to retrieve all fonts and images from the disk in this measurement process.

Because this procedure disadvantages TeX engines by requiring them to read a file from disk (and by measuring the overhead to create and write that file), we apply a correction factor for each transaction. We calculate it by writing files for the source state after each action to the disk, passing them to the `cat` command to read them and receive the standard output of `cat` in the benchmark binary. In the reported times for TeX engines, we subtract this correction factor from the measured runtime.

In some other aspects, this setup is favorable for LaTeX: Typst's incremental compilation is designed to be used together with its web app, directly rendering the frames to a screen buffer. Instead, to perform comparable tasks in these benchmarks, Typst is used in the way one would typically use LaTeX: A PDF is generated from the output frames using Typst's PDF exporter. This component is not subject to any caching or incremental computing. Some tasks like embedding and subsetting fonts have to be performed for all documents, regardless of cache hotness or document size, imposing a lower runtime bound on Typst's PDF export.

Because, as mentioned in the previous subsection, wall clock or CPU time benchmarks can be quite noisy, we took precautions to obtain meaningful results. The runtime for each document/transaction/engine combination was measured independently at least ten times. In each of the measurements, our benchmark runner executed the transaction multiple times if it took less than three seconds to execute a single time. Then, the reported time is the measured time divided by the number of transaction executions. Before each measurement series, the document/transaction/engine combination was run at least once to warm up the hardware caches of the system.

The benchmarks were run on a Microsoft Surface Book 2 13″ with an Intel Core i7-8650U and 16GB of RAM. All energy savings modes were turned off. The test executables were GNU/Linux binaries executed on the Windows Subsystem for Linux 2 (WSL2) on Debian 9 and with an `ext4` filesystem. The WSL2 system was hosted by Windows 11 (build 22000).

	Insertion		Deletion	
pdfTeX	($\sigma = 05.85$)	241.49	($\sigma = 14.03$)	197.57
XeTeX	($\sigma = 30.94$)	814.98	($\sigma = 18.70$)	767.06
LuaTeX	($\sigma = 33.67$)	1044.54	($\sigma = 30.60$)	1069.48
Typst (clean)	($\sigma = 00.06$)	9.51	($\sigma = 00.11$)	9.80
Typst (incr.)	($\sigma = 00.00$)	0.10	($\sigma = 00.00$)	0.11

Work Sheet

	Insertion		Deletion	
pdfTeX	($\sigma = 053.84$)	2968.52	($\sigma = 112.78$)	3100.19
XeTeX	($\sigma = 478.63$)	12894.31	($\sigma = 234.09$)	12475.32
LuaTeX	($\sigma = 133.00$)	4046.07	($\sigma = 138.47$)	3989.62
Typst (clean)	($\sigma = 000.09$)	10.54	($\sigma = 000.08$)	10.67
Typst (incr.)	($\sigma = 000.00$)	0.30	($\sigma = 000.01$)	0.36

Brochure

	Insertion		Deletion	
pdfTeX	($\sigma = 060.47$)	2218.60	($\sigma = 071.60$)	2142.24
XeTeX	($\sigma = 066.67$)	4012.01	($\sigma = 062.69$)	3992.86
LuaTeX	($\sigma = 219.54$)	14507.59	($\sigma = 165.01$)	13974.35
Typst (clean)	($\sigma = 079.96$)	2987.18	($\sigma = 048.45$)	2904.41
Typst (incr.)	($\sigma = 019.54$)	657.30	($\sigma = 033.26$)	632.17

Tome

Table 6: Comparison of TeX engines and Typst with mean time in milliseconds and standard deviation for recompilation after a single editing action.

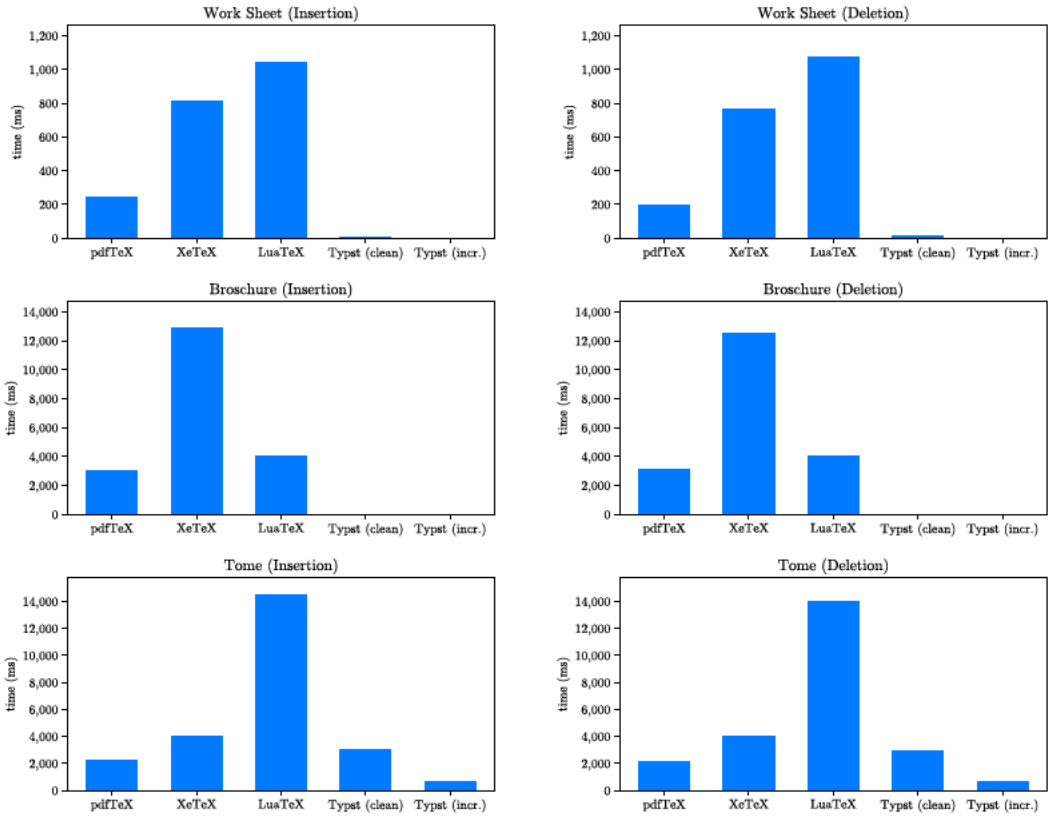


Figure 17: Average compile time for a PDF with a single editing action in milliseconds.

The first takeaway from these measurements is that Typst, except for very large documents, is already an order of magnitude faster than the fastest measured LaTeX engine, pdfTeX. The incremental compilation procedure can then increase the performance by another factor, between 4.5x (Tome, Insertion) and 95.1x (Work Sheet, Deletion). Ultimately this increase leads to the result that, with incremental compilation, Typst is between 3.4x (Tome, Insertion) and 9,895x (Brochure, Insertion) faster than the fastest TeX engine.

The measurements show that the old pdfTeX engine is always the fastest. Amongst the two engines that support OpenType fonts, LuaTeX and XeTeX, the former is slower, except for the brochure document with many packages and a custom font. The approach of LuaTeX is to implement as much as possible in Lua (e.g., the shaper and font tools) and expose it to the user for high customizability. In contrast, XeTeX will link against the operating system's native text libraries like Core Text and DirectWrite and use HarfBuzz as a shaper. The slow XeTeX performance for the brochure could be explained by either slow I/O and parsing routines (impactful because many packages are loaded), slow image en- and decoding (Brochure contains the largest images, Tome none), or inadequate performance of the Linux text libraries.

The speedup observed in these measurements for Typst with incremental compilation is lower than that observed in Table 5 because this benchmark is the first to include the PDF exporter. Typst's exporter will re-encode each image in each compilation to embed it in the file when exporting PDFs, even if it could have pasted the original encoded file into the PDF file. This fact explains the discrepancies with the brochure. Also, for writing large files, the export becomes a significant part of the runtime again, explaining the slowdowns observed with Tome.

7 Future Work

This thesis makes two of the four stages of the Typst compiler incremental; Evaluation and Export remain non-incremental and have to redo everything in each compilation. Future work should focus on enabling the reuse of past compilation results for these stages as well, leveraging the work done in this thesis. Ultimately, the goal here should be to provide sub-linear compilation times in terms of overall file size for the average edit.

In the Evaluation stage, the stability of the AST between compilations afforded by incremental parsing could be used. This stability enables an incrementalized evaluation to know which nodes changed without a diff. Such an algorithm would not be trivial because it would still need to consider the interdependencies like identifier references that are semantical but not syntactical.

Because Typst is used with multiple exporters (PDF, raster images, …), it would be worthwhile to research a generalized approach for incremental export. An eventual incremental exporting algorithm that interfaces with the layout cache should be architected such that any exporter does not have to provide a manual, independant implementation. In such a scheme, both data about the reusability of artifacts and the caching mechanism could be provided in the shared part of Typst while remaining agnostic about the export target, allowing more rapid implementation of and iteration on exporters without duplicate caching implementations.

The incremental parser could be enhanced by performing a rigorous analysis of the classes of languages it accepts and by mathematically proving its correctness for them. Which effect a token must have to be considered in the lookback search could also be more thoroughly investigated.

The worst-case runtime complexity of the incremental parser could be lowered by introducing intermediate nodes in the green tree that prevent a node from having too many children and thus the tree from being too flat and wide. Backshifting the list of nodes when deleting at the start causes much work in such trees. These nodes would need to consider the unique syntactic properties of the language being parsed because their first and last children would always constitute a valid reparse boundary. Intermediate nodes must be made invisible in the syntax tree to retain the developer experience of the Red-Green-Tree.

A considerable challenge for implementors of the layout cache proposed in this thesis is manually determining a scheme to derive constraints for each available layouter. This is error-prone and laborious when introducing new layouters. A worthwhile future extension of the constrained layout cache would be automatically generating the constraints based on the comparisons the layouter has performed on the Region sizes it must use for layouting.

It would also be interesting to see a more extensive investigation of the performance characteristics of the constrained layout cache versus a browser-like two-stage layout approach with size measurement and positioning phases in an otherwise identical system.

8 Conclusion

In this thesis, we tackled how to increase the performance of a markup-based typesetting system with incremental compilation fast enough for users to perceive its recompilations as interactive for all typical documents and edits. We chose the incremental compilation approach because most edits to a typical document are minor, and users expect previews to refresh after each keystroke. The aim was to "[ensure] that the time to respond to small changes is proportionately small," (Baron 2003) just as users expect from their experience with non-compiler based WYSIWYG text processing and typesetting software.

We conducted the research in this thesis based on Typst, a novel markup-based typesetting system that the author is involved in creating. Typst has four compilation stages: Parsing, evaluation, layouting, and export. This thesis "incrementalizes" two of them: Layouting, the longest-running stage of all of them, and parsing, because it is vital for the responsiveness of syntax highlighting and becomes significant for large documents.

In this thesis, we have presented a novel approach for refitting existing handwritten recursive descent parsers based on Red-Green-Trees to exhibit incremental characteristics. Nodes affected by a change are located in their position-independent subtree, and a reparsing range is computed based on language characteristics. Reparsing continues until convergence with the prior tree occurs, accomodating even some context-sensitive grammars. This approach is especially suited to replacing some nodes of a Concrete Syntax Tree with a flat hierarchy (e.g., some words in markup). We could not use an existing algorithm because of the context-sensitive nature of Typst markup and the need to retain the handwritten recursive descent parser to provide user-friendly error messages.

The second contribution of this thesis is a constraint-based layout caching scheme, complete with a custom cache eviction policy. The caching scheme is based on the fact that when the compiler requests some elements to be placed within allocated space on the page, the outcome might be the same between two passes, even if the width or height of the allocated space slightly changes. Each layout result is thus annotated with constraints describing space allocations under which it may be reused. The thesis contains instructions on how to derive such constraints for various layouters. Multiple layout results may be cached for the same element. The eviction scheme enhances the Least Frequently Used policy with more fine-grained data about recent use, allowing the application of heuristics to better align eviction with users' performance expectations for a typesetting system.

We validate both approaches with a large corpus of test cases. For parsing, random changes are automatically introduced, prioritizing locations in the source file that can have a high impact on the parse tree structure. The changed document is then parsed incrementally and from scratch, comparing the results. For layouting, cache entries are checked for the plausibility of their constraints given the original regions. The corpus is also used to layout each document multiple times, discarding cached results for all but one layer of the layout tree and comparing them against the clean original layout.

We evaluated both techniques alone and in tandem by running them in a CPU and memory simulation in Cachegrind to eliminate any benchmark noise caused by background tasks in the system or the non-deterministic behavior of the CPU. We performed these measurements with a matrix of 3 documents and edits each.

We show that incremental parsing improves performance by a factor between 3.72

and 19.73, and the layout cache improves performance by factors between 4.80 and 83.96. Both modifications combined speed up the Typst compiler by between 4.56 and 80 times.

We also compared a clean, full compilation and incremental compilation of a Typst source file and its edits against various TEX compilers in a wall-clock benchmark. We found that, depending on the document's characteristics, the incremental Typst compiler is between 3.4 and 9895 times faster than the fastest TEX engine. The incremental compilation techniques help Typst reach an interactive 30 and 60 frames per second for more documents.

Overall, this thesis has shown that compiler and markup-based typesetting systems can compete with the performance of WYSIWYG solutions, in particular, when using intelligent caching strategies that leverage the small size of source changes between compilations. The benefit of allowing a long-running compiler to memorize compilation artifacts is tangible and the key to unlocking this class of performance, even if text shaping, reordering, and layouting would otherwise be prohibitively slow for this purpose. This thesis shows an avenue for LATEX users, in particular, to unshackle themselves from poor user experience because it demonstrates that a markup-based typesetting system can be fast and provide readable error messages while still providing high-quality output.

Finally, it may be noted that this thesis is set in Typst, and was edited using the technologies presented here.

References

Abrahams, Paul, Kathryn Hargreaves, and Karl Berry. 1990. *TEX for the Impatient.* Amsterdam: Addison Wesley Longman, https://tug.ctan.org/info/impatient/book.pdf.

Aliprand, Joan M. 2011. "The Unicode Standard." *Library Resources & Technical Services* 44, no. 3, 160–167. https://doi.org/10.5860/lrts.44n3.160.

Alison, Nat et al. 2021. "Reconciliation," accessed May 16, 2022, https://reactjs.org/docs/reconciliation.html.

Allard, J., and B. Raffin. 2005. "A Shader-Based Parallel Rendering Framework." In *VIS 05. IEEE Visualization, 2005.* Conference Presentation at VIS 05. IEEE Visualization, 2005., 127–134, https://doi.org/10.1109/VISUAL.2005.1532787.

Ama et al. 2020. "Writer/core and Layout," accessed May 17, 2022, https://wiki.openoffice.org/w/index.php?title=Writer/Core_And_Layout&oldid=239359.

Anderson, Brian et al. 2016. "Engineering the Servo Web Browser Engine Using Rust." In *Proceedings of the 38th International Conference on Software Engineering Companion.* New York, NY, USA: Association for Computing Machinery, 81–89, https://doi.org/10.1145/2889160.2889229.

Apple Developer. 2018. *Model-View-Controller.* Cupertino, CA, USA: Apple, accessed May 16, 2022, https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html.

Atkin, Steve, and Ryan Stansifer. 2000. *Implementations of Bidirectional Reordering Algorithms.* Melbourne, FL, USA: Florida Tech, 10, accessed May 21, 2022, https://cs.fit.edu/media/TechnicalReports/cs-2000-1.pdf.

Atkins, Tab Jr. 2022. *CSS Will Change Module Level 1.* W3C, accessed May 25, 2022, https://www.w3.org/TR/2022/CRD-css-will-change-1-20220505/.

Aycock, John. 2001. "Why Bison Is Becoming Extinct." *XRDS: Crossroads, the ACM Magazine for Students* 7, no. 5, 3. https://doi.org/10.1145/969637.969640.

Baron, David. 2003. *Cleaning Up Layout,* accessed May 16, 2022, https://groups.google.com/g/netscape.public.mozilla.layout/c/Vo3-EvIhELs/m/PRSqCoLEtAsJ?pli=1.

Barth, Adam. 2016. *Flutter's Rendering Pipeline.* Presentation at Google Tech Talks, Cupertino, CA, USA. Accessed May 16, 2022, https://www.youtube.com/watch?v=UUfXWzp0-DU.

Beebe, Nelson HF. 2004. "25 Years of TEX and METAFONT: Looking Back and Looking Forward TUG 2003 Keynote Address." *TUGBoat* 25, no. 1, 7–30. https://www.tug.org/TUGboat/Articles/tb25-1/beebe-2003keynote.pdf.

Bouch, Anna, Allan Kuchinsky, and Nina Bhatti. 2000. "Quality Is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* New York, NY, USA: Association for Computing Machinery, 297–304, https://doi.org/10.1145/332040.332447.

Breitenlohner, Peter. 1998. *The ε-TEX Manual.* München, Germany: Max-Planck-Institut, 20, accessed September 30, 2021, https://ftp.rrze.uni-erlangen.de/ctan/systems/doc/etex/etex_man.pdf.

Brunsfeld, Max. 2021. *Tree-Sitter,* accessed October 1, 2021, https://tree-sitter.github.io/tree-sitter/.

Brunsfeld, Max. 2018. *Tree-Sitter - a New Parsing System for Programming Tools.* Presentation at Strange Loop, St. Louis, MO, USA. Accessed October 3, 2021, https://www.thestrangeloop.com/2018/tree-sitter---a-new-parsing-system-for-programming-tools.html.

Chaffraix, Julien et al. 2022. *Layout_object.h.* Mountain View, CA, US, accessed May 16, 2022, https://source.chromium.org/chromium/chromium/src/+/main:third_party/blink/renderer/core/layout/layout_object.h;l=184;drc=f7e7475f2900842a1474f6c1bc368c76b9a5d3c0.

Chapman, Christopher. 2021. *UAX #14: Unicode Line Breaking Algorithm.* The Unicode Consortium, accessed November 22, 2021, https://unicode.org/reports/tr14/.

cwzwarich, and Eric Lippert. 2020. "Does the C# Parser Implement Incremental Parsing by Just Using a Recursive Descent Parser," accessed May 13, 2022, https://news.ycombinator.com/item?id=22712676.

Dubroy, Patrick, and Alessandro Warth. 2017. "Incremental Packrat Parsing." In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering.* New York, NY, USA: Association for Computing Machinery, 14–25, https://doi.org/10.1145/3136014.3136022.

Eaton, Phil. 2021. "Parser Generators Vs. Handwritten Parsers: Surveying Major Language Implementations in 2021," accessed October 3, 2021, https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html.

Edge. 2012. "Denial of Service Via Hash Collisions." *Lwn.net.* https://lwn.net/Articles/474912/.

Eigner, Kai. 2017. "Using Harfbuzz as Opentype Engine in Luatex." *Maps* 47. https://www.ntg.nl/maps/47/02.pdf.

Enlyft. 2022. "Desktop Publishing Products," accessed May 30, 2022, https://enlyft.com/tech/desktop-publishing.

Erdweg, Sebastian et al. 2013. "Layout-Sensitive Generalized Parsing." In *Software Language Engineering.* Eds. Krzysztof Czarnecki and Görel Hedin. Berlin, Heidelberg: Springer, 244–263, https://doi.org/10.1007/978-3-642-36089-3_14.

Felici, James. 2012. *The Complete Manual of Typography.* 2nd ed. Berkeley, CA: Peachpit.

Ford, Bryan. 2004. "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation." In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA: Association for Computing Machinery, 111–122, https://doi.org/10.1145/964001.964011.

Fu, Yupeng et al. 2010. "Ajax-Based Report Pages as Incrementally Rendered Views." In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data.* New York, NY, USA: Association for Computing Machinery, 567–578, https://doi.org/10.1145/1807167.1807229.

Gafter, Neal M. 1990. *Parallel Incremental Compilation.* Thesis, Rochester, NY: University of Rochester, accessed May 12, 2022, https://apps.dtic.mil/sti/pdfs/ADA228726.pdf.

Garland, A. C. et al. 2020. *Swift Syntax and Structured Editing Library,* accessed October 1, 2021, https://github.com/apple/swift/blob/main/lib/Syntax/README.md.

Ghezzi, Carlo, and Dino Mandrioli. 1979. "Incremental Parsing." *ACM Transactions on Programming Languages and Systems* 1, no. 1, 58–70. https://doi.org/10.1145/357062.357066.

Goossens, Michel, ed. 2010. *The* XeTeX *Companion.* Genf, Switzerland: CERN, 112, https://xml.web.cern.ch/lgc2/xetexmain.pdf.

Gruber, John. 2004. "Markdown Syntax Documentation," accessed November 11, 2021, https://daringfireball.net/projects/markdown/syntax.

Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation.* 3rd ed. Boston: Pearson.

Jalili, Fahimeh, and Jean H. Gallier. 1982. “Building Friendly Parsers.” In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA: Association for Computing Machinery, 196–206, https://doi.org/10.1145/582153.582175.

Jeffery, Clinton L. 2003. “Generating LR Syntax Error Messages from Examples.” *ACM Transactions on Programming Languages and Systems* 25, no. 5, 631–640. https://doi.org/10.1145/937563.937566.

Biber: A Backend Bibliography Processor for Biblatex. 2022. Philip Kime and François Charette, 57, accessed May 18, 2022, https://ftp.rrzn.uni-hannover.de/pub/mirror/tex-archive/biblio/biber/documentation/biber.pdf.

Kladov, Aleksey et al. 2022. *Rust-Analyzer: Reparsing.rs,* accessed May 13, 2022, https://github.com/rust-lang/rust-analyzer/blob/0c881e882e11137a74e1b185609b9056df0282c1/crates/syntax/src/parsing/reparsing.rs.

Kladov, Aleksey et al. 2021. *Syntax in Rust-Analyzer.* rust-analyzer, accessed October 1, 2021, https://github.com/rust-analyzer/rust-analyzer/blob/8b45de596f06646b13ed0fdc8b7a80a271f3b174/docs/dev/syntax.md.

Knuth, Donald E., and Michael F. Plass. 1981. “Breaking Paragraphs into Lines.” *Software: Practice and Experience* 11, no. 11, 1119–1184. https://doi.org/10.1002/spe.4380111102.

Knuth, Donald E. 1986. *The TEXbook.* 6th ed.. Computers & Typesetting. Reading, MA: Addison Wesley.

Tex, initex - Text Formatting and Typesetting. 2020. TEXLive2020. Donald E. Knuth et al., accessed September 30, 2021, https://manpages.ubuntu.com/manpages/impish/man1/tex.1.html.

Knuth, Donald E. 1965. “On the Translation of Languages from Left to Right.” *Information and Control* 8, no. 6, 607–639. https://doi.org/10.1016/S0019-9958(65)90426-2.

Lagally, Klaus. 2004. *Arabtex: Typesetting Arabic and Hebrew.* Stuttgart, Germany: Universität Stuttgart, accessed May 18, 2022, http://baobab.informatik.uni-stuttgart.de/arabtex/doc/arabdoc.pdf.

Lamport, Leslie. 1986. *LATEX: A Document Preparation System.* 13th ed. Reading, MA: Addison Wesley.

Landin, P. J. 1966. “The Next 700 Programming Languages.” *Communications of the ACM* 9, no. 3, 157–166. https://doi.org/10.1145/365230.365257.

Lang, Bernard. 1974. “Deterministic Techniques for Efficient Non-Deterministic Parsers.” In *Automata, Languages and Programming, 2nd Colloquium.* Saarbrücken: Springer, 255–269, https://doi.org/10.1007/3-540-06841-4_65.

Lev, Noah et al. 2022. “Queries: Demand-Driven Compilation,” accessed May 24, 2022, https://rustc-dev-guide.rust-lang.org/query.html.

Lewis, Paul, and Surma. 2018. “CSS Triggers,” accessed September 30, 2021, https://csstriggers.com/.

Lewis, Paul. 2021. “Rendering Performance,” accessed September 30, 2021, https://developers.google.com/web/fundamentals/performance/rendering.

Linton, M.A., J.M. Vlissides, and P.R. Calder. 1989. “Composing User Interfaces with Interviews.” *Computer* 22, no. 2, 8–22. https://doi.org/10.1109/2.19829.

Lippert, Eric. 2012. “Persistence, Façades and Roslyn’s Red-Green Trees,” accessed October 1, 2021, https://ericlippert.com/2012/06/08/red-green-trees/.

LuaTEX development team. 2022. *LuaTEX Reference Manual,* 324, accessed May 18, 2022, https://ctan.mirror.norbert-ruehl.de/systems/doc/luatex/luatex.pdf.

Mädje, Laurenz. Forthcoming. *Typst: A Programmable Markup Language for Typesetting.* Thesis, Berlin: Technische Universität Berlin.

Matsakis, Niko et al. 2020. *Rustc-hash,* accessed June 12, 2022, https://github.com/rust-lang/rustc-hash.

MDN contributors. 2021. "Populating the Page: How Browsers Work," accessed September 30, 2021, https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work.

Mittelbach, Frank. 2014. "How to Influence the Position of Float Environments Like Figure and Table in LaTeX?" *TUGBoat* 35, no. 3, 248–254. https://www.latex-project.org/publications/2014-FMi-TUB-tb111mitt-float-placement.pdf.

Nilsson, Peter, and David Reveman. 2004. "Glitz: Hardware Accelerated Image Compositing Using Opengl." In *Proceedings of the 2004 USENIX Annual Technical Conference,* https://www.usenix.org/legacy/event/usenix04/tech/freenix/full_papers/nilsson/nilsson_html/.

Ollila, Risto. 2021. *A Performance Comparison of Rendering Strategies in Open Source Web Frontend Frameworks.* Thesis, Helsinki, Finland: University of Helsinki, accessed May 16, 2022, https://helda.helsinki.fi/bitstream/handle/10138/329793/Ollila_Risto_tutkielma_2021.pdf?sequence=2&isAllowed=y.

Palumbo, Daniele. 2021. *The Flutter Framework: Analysis in a Mobile Enterprise Environment.* Thesis, Turin, IT: Politecnico di Torino, accessed May 17, 2022, https://webthesis.biblio.polito.it/19111/1/tesi.pdf.

Panchekha, Pavel, and Chris Harrelson. 2021. "Saving Partial Layouts," accessed May 16, 2022, https://www.browser.engineering/reflow.html.

Parsons, Jared et al. 2022. *Roslyn Blender Library,* accessed May 13, 2022, https://github.com/dotnet/roslyn/blob/1cca63b5d8ea170f8d8e88e1574aa3ebe354c23b/src/Compilers/CSharp/Portable/Parser/Blender.cs.

Python Software Foundation. 2021. *2. Lexical Analysis,* accessed October 3, 2021, https://docs.python.org/3/reference/lexical_analysis.html.

Richardson, Leland. 2019. "Compose from First Principles," accessed May 17, 2022, http://intelligiblebabble.com/compose-from-first-principles/.

Rohou, Erven, Bharath Narasimha Swamy, and André Seznec. 2015. "Branch Prediction and the Performance of Interpreters - Don't Trust Folklore." In . Conference Presentation at International Symposium on Code Generation and Optimization, 12, https://doi.org/10.1109/CGO.2015.7054191.

Rosenkrantz, D. J., and R. E. Stearns. 1970. "Properties of Deterministic Top-Down Grammars." *Information and Control* 17, no. 3, 226–256. https://doi.org/10.1016/S0019-9958(70)90446-8.

Silberschatz, Abraham, Peter Baer Galvin, and Greg Gagne. 2008. *Operating System Concepts.* 8th ed. Hoboken, N. J: John Wiley & Sons.

Sippu, Seppo, and Eljas Soisalon-Soininen. 1990. *Parsing Theory: Volume II LR(K) and LL(K) Parsing.* Soft cover reprint of the original 1st ed. Berlin; London: Springer.

SQLite. 2019. "Measuring and Reducing CPU Usage in Sqlite," accessed March 23, 2022, https://www.sqlite.org/cpu.html.

Syromiatnikov, Artem, and Danny Weyns. 2014. “A Journey Through the Land of Model-View-Design Patterns.” In *2014 IEEE/IFIP Conference on Software Architecture.* Conference Presentation at 2014 IEEE/IFIP Conference on Software Architecture, 21–30, https://doi.org/10.1109/WICSA.2014.13.

Brink, Alex ten, and Antonio Maniero. 2010. “Language Design - Should I Use a Parser Generator or Should I Roll My Own Custom Lexer and Parser Code?,” accessed October 3, 2021, https://softwareengineering.stackexchange.com/questions/17824/should-i-use-a-parser-generator-or-should-i-roll-my-own-custom-lexer-and-parser.

Thành, Hàn Thế. 2000. *Micro-Typographic Extensions to the TEX Typesetting System.* Thesis, Brno, Czech Republic: Masaryk University, accessed September 30, 2021, https://www.pragma-ade.nl/pdftex/thesis.pdf.

The pdfTEX User Manual. 2021. Hàn Thế Thành et al., 55, accessed September 30, 2021, https://ctan.org/pkg/pdftex.

The Clang Team. 2022. *Modules — Clang 15.0.0git Documentation,* accessed May 18, 2022, https://clang.llvm.org/docs/Modules.html.

The Kermit Project. 2019. “The DEC La36 Decwriter II Terminal,” accessed December 7, 2019, http://www.columbia.edu/kermit/terminals.html.

Tomita, Masaru. 1991. *Generalized LR Parsing.* 1991th ed. Boston: Springer.

Turner-Trauring, Itamar. 2020. “CI for Performance: Reliable Benchmarking in Noisy Environments,” accessed March 23, 2022, https://pythonspeed.com/articles/consistent-benchmarking-in-ci/.

tyoverby. 2017. “Hello, I Work on the C# Compiler and We Use a Handwritten Recursive-Descent Parser,” accessed April 26, 2022, https://news.ycombinator.com/item?id=13915150.

Vajna, Miklos. 2014. *Writer Training #1, 2014.* Online Presentation. Accessed May 17, 2022, https://speakerdeck.com/vmiklos/writer-training-number-1-2014.

Rossum, Guido van, Pablo Galindo, and Lysandros Nikolaou. 2020. *New PEG Parser for CPython.* Python Software Foundation, accessed April 26, 2022, https://peps.python.org/pep-0617/.

Vasilijević, Vasilije, Nenad Kojić, and Natalija Vugdelija. 2020. “A New Approach in Quantifying User Experience in Web-Oriented Applications.” In *4th International Scientific Conference on Recent Advances in Information Technology, Tourism, Economics, Management and Agriculture.* Belgrade: Association of Economists and Managers of the Balkans, https://doi.org/10.31410/ITEMA.2020.9.

Wagner, Tim A. 1998. *Practical Algorithms for Incremental Software Development Environments.* Thesis, Berkeley, CA, USA: EECS Department, University of California, Berkeley, accessed October 1, 2021, https://www2.eecs.berkeley.edu/Pubs/TechRpts/1997/5885.html.

Woerister, Michael. 2016. “Incremental Compilation,” accessed October 1, 2021, https://blog.rust-lang.org/2016/09/08/incremental.html.

Yaakov. 2017. “Red-Green Trees,” accessed October 1, 2021, https://blog.yaakov.online/red-green-trees/.

Yedidia, Zachary, and Stephen Chong. 2021. “Fast Incremental PEG Parsing.” In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering.* New York, NY, USA: Association for Computing Machinery, 99–112, https://doi.org/10.1145/3486608.3486900.

Yergeau, François. 2003. *UTF-8, a Transformation Format of ISO 10646.* Network Working Group, 14, accessed May 18, 2022, https://www.hjp.at/doc/rfc/rfc3629.html.

Zhang, Kaimin et al. 2010. “Smart Caching for Web Browsers.” In *Proceedings of the 19th International Conference on World Wide Web*. New York, NY, USA: Association for Computing Machinery, 491–500, https://doi.org/10.1145/1772690.1772741.

The images in Listing 6 are taken by Vlad Tchompalov, Shirish Suwal, Jeremy Hynes, and Jason Leung (left to right, top to bottom) and are obtained from the Unsplash library. All of them are in the public domain or permissively licensed.

Appendix

Code for the Insertion Transaction:

```
// https://en.wikipedia.org/wiki/Jessie_Craigen
It is not certain where in the UK she was born, although the Dover Express
in 1866 described her as a 'Scotch lady'. The 1871 census, however, shows
her living with an adopted 18-year-old child, Rosetta Vincent, and a married
sister, Emma Henley, in Ordsall near Retford and describing herself as a
‘lecturer’,
born in London. By 1881 she is in Clifton, Bristol, and in the census she
describes herself as a London-born, ‘Lecturer on Social Subjects’.

She reportedly had a seafaring father from the Scottish Highlands,
who died when she was an infant, and a mother who was an Italian actress.
As a child she appeared on the stage and this may have given her the skills
and the confidence for paid public speaking.
```

Code for the Grid Transaction; also compare with Figure 15:

```
#grid(
  columns: (1fr, 3cm, 2fr, auto),
  gutter: 6pt,

  rect(width: 100%, padding: 6pt)[A rectangle with some text could be here.],
  [
    Lorem ipsum dolor sit amet,
    sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
  ],
  rect(
    width: 100%,
    fill: maroon,
    stroke: black,
    padding: 8pt,
    align(center, circle(radius: 20pt, stroke: white, thickness: 3pt))
  ),
  rect(
    width: 5cm,
    padding: 6pt,
    for i in range(8) {rect(width: 20%, fill: rgb(100%, 100% * (i / 8), 0%))}
  ),
  [
    #set par(lang: "de")
    Meine Oma fährt im Hühnerstall Motorrad.
  ],
  rect(width: 100%),
  align(center)[*Vexillology*],
  align(bottom+right, square(width: 4pt, fill:black))
)
```