# The Superuser Guide to LaTeX

Flyxion

*Independent Researcher*

2026

# Preface

This is not an introductory LaTeX manual. Introductions to LaTeX are plentiful, well written, and not the gap this book is trying to fill. Most of them teach a reader how to produce a correct document: how to start a section, insert a table, number an equation, cite a source. That is necessary knowledge, and this book assumes a reader already has it, at least in outline. What that kind of introduction does not usually teach, because it is a different kind of knowledge entirely, is how to design and maintain a LaTeX project that has grown past the scale a single file, a weekend, or a single author's unaided memory can comfortably hold.

This book exists for that second kind of knowledge. Its central claim, argued from several different directions across twenty-two chapters, is that LaTeX is best treated not as a typesetting language to be learned once and then simply used, but as a programmable, reproducible publishing environment, closer in spirit to a software project than to a word-processed document. Once that reframing is taken seriously, a great deal of what otherwise looks like idiosyncratic LaTeX trivia, why a build needs two passes, why a fragile command breaks only sometimes, why a five-hundred-page manuscript should never live in one file, turns out to be the direct, predictable consequence of a small number of underlying principles, the same principles a software engineer would recognize from any other large, long-lived, collaboratively maintained system.

The book is organized to build that understanding from the ground up: how TeX actually processes a document before any LaTeX convention is layered on top of it; how to architect, version, and build a large multi-file project; how to write macros and interfaces that behave predictably; how to use modern font technology and mathematical typesetting to their full extent; how to manage bibliographies, indexes, and glossaries at real scale; how to bring graphics and computation directly into the document's own build process rather than pasting in artifacts prepared elsewhere; and, in its final chapters, how to think about an entire body of published work, spanning years and multiple books, as one maintained, coherent system rather than a shelf of unrelated documents.

Two companion volumes sit alongside this one without being required reading for it. *Thinking in Vim* is about editing structured text efficiently, treating an editor as more than a place to type. *Thinking Like Rust* is about building computation reliably, with the kind of explicit, deterministic discipline that a document's own correctness can depend on when a figure or a table is generated rather than hand-copied. Each book stands on its own, and nothing in this one assumes the other two have been read. But a reader who has encountered all three will recognize the same underlying commitment running through each: that a technical practice, whether editing, computing, or publishing, rewards being

learned as a designed system rather than picked up as a bag of separately memorized tricks. This book is the last of the three stages, the one responsible for taking whatever was written and computed and turning it into something durable enough to be read.

# Acknowledgements

This book, like the rest of the corpus it belongs to, was developed through iterative drafting and revision rather than in a single sustained sitting, and it owes whatever clarity it has to that process of repeated reworking rather than to any single moment of insight. Readers of earlier drafts of the material collected here, and collaborators on the broader research program this book's conventions were drawn from, have the author's thanks, even where they are not named individually in this edition.

# Who This Book Is For

This book is for a reader who already knows how to write a LaTeX document and now wants to know how to build one that lasts. That includes, without being limited to: an academic or independent researcher maintaining a long-running body of technical writing across many documents; a graduate student or postdoc whose thesis or first book-length manuscript has outgrown the single-file habits that worked fine for a conference paper; a software engineer who has picked up LaTeX for documentation or papers and wants to apply the same engineering discipline to it that they would apply to a codebase; and anyone maintaining a technical book, textbook, or reference work expected to remain buildable, correct, and internally consistent for years after it is first written.

It is not primarily for a reader who has never used LaTeX before, though such a reader is welcome to use this book as a map of where their skills might eventually lead, alongside a more conventional introduction covering the syntax this book largely assumes. It is also not a reference manual for any single package; each chapter points to the specific tools relevant to its subject, but the authoritative documentation for any given package remains the right place to look up that package's full interface. What this book offers instead is the architecture around those tools: how to choose among them, how to combine them into a working system, and how to keep that system working as it, and the body of work built with it, continues to grow.

# Contents

# Part I

# Foundations

# Chapter 1

# Thinking Like TeX

Most introductions to LaTeX begin with commands: how to start a section, how to insert a table, how to number an equation. This book begins somewhere else, because commands are the wrong level of description for a superuser. A LaTeX document is not a sequence of instructions given to a word processor; it is the output of a small, precisely specified computation carried out by TeX itself, with LaTeX supplying the vocabulary in which that computation is expressed. Understanding what TeX actually does with a document, rather than what LaTeX appears to do with it, is the difference between a user who works around the engine's behavior by trial and error and one who predicts it.

## 1.1 Boxes, Glue, and Penalties

TeX's model of a typeset page is built from three primitive kinds of material: boxes, glue, and penalties. A box is a rectangular region of fixed width, height, and depth, containing either a piece of typeset text, a rule, or other boxes nested inside it. Every character TeX sets is, at some level, wrapped in a box; every word is a horizontal sequence of character boxes; every line is a horizontal box containing those words; every paragraph is a vertical stack of lines, itself a vertical box.

Boxes alone would produce rigid, unadjustable pages, since a box has a single fixed size. TeX solves this with glue: a flexible space with a natural width and the capacity to stretch or shrink by specified amounts under specified conditions. The space between words is glue, not a fixed-width character; when TeX justifies a line, it is not inserting or deleting spaces but instructing the glue between words to stretch or shrink until the line fills its target width. Vertical space between paragraphs, before and after section headings, and around floats is glue for exactly the same reason: it needs to absorb the differences between one page and the next without being edited by hand.

Penalties are TeX's mechanism for expressing a preference without imposing a rule. A penalty is a number attached to a potential break point in a line or a page; large positive penalties discourage TeX from breaking there, large negative penalties encourage it, and a penalty of $-10000$ or more forces a break outright. The widow and orphan control built into LaTeX's paragraph and page-breaking behavior, along with commands like `\nopagebreak` and `\penalty`, are all instructions expressed in this currency.

**Definition.** A *layout object* in TeX is either a box (fixed extent, opaque content), a glob of glue (an extent with stretch and shrink components), or a penalty (a signed integer attached to a candidate break point). A horizontal or vertical list is a sequence of layout objects; typesetting a paragraph or a page is the process of choosing a subset of the penalties in a list to act as actual breaks, subject to the constraint that the glue in each resulting segment can stretch or shrink to the required target size.

This is worth stating plainly because it explains behavior that otherwise looks like a bug. A paragraph that refuses to justify without visibly stretched spacing has run out of glue to stretch; a section heading that jumps to the next page did so because a penalty forbade breaking immediately after it; a table that will not sit where it was written moved because floats are boxes subject to their own placement rules, not because LaTeX is arbitrarily uncooperative. Once boxes, glue, and penalties are visible as the actual material TeX is working with, most of what look like idiosyncrasies of the engine turn out to be the direct, legible consequence of a small set of rules applied consistently.

## 1.2 Tokens, Category Codes, and the Lexer/Expander Split

Before any of that layout machinery runs, TeX has to turn a stream of characters in a source file into something it can act on. This happens in two stages that are worth keeping separate in one's head, because most confusing macro behavior comes from conflating them.

The first stage is tokenization. TeX reads raw characters and, using a table of *category codes*, converts each one into a token: a control sequence (`\section`, `\newcommand`), a character with a category such as letter, space, math shift, or alignment tab, or one of a small number of special tokens. Category codes are not fixed properties of characters; they are assignments that can be changed. The fact that `#` means "parameter marker" inside a macro definition but a literal hash character when its category code is altered, or that `%` can be made to stop being a comment character, follows from the same mechanism: TeX does not know what a character "means" until it has consulted the category code table current at the moment that character is read.

The second stage is expansion. An expandable token, such as a macro name, is repeatedly replaced by its definition until only unexpandable tokens remain: primitive commands, characters ready to be typeset, or explicit instructions like box and glue specifications. This is the stage most macro-writing takes place in, and it is where the rewriting-system character of TeX becomes visible. Defining a macro with `\newcommand` or `\def` does not create a procedure that TeX will later call; it registers a rewrite rule that TeX applies wherever the macro's name appears, exactly as many times as expansion requires and no more.

Keeping tokenization and expansion apart clarifies a class of problems that otherwise look mysterious: a macro that behaves differently depending on what character follows it, a command that fails only inside a `verbatim` environment, or a package that changes the meaning of an ordinary character such as `"` or `~`. In each case, the category code table active at the point a character is read has changed, and the tokens produced from the same visual character sequence are simply different tokens than they would otherwise have been.

## 1.3 Expansion Order: What TeX Sees versus What TeX Does

A recurring source of difficulty for LaTeX users moving from ordinary commands to writing their own macros is that expansion does not proceed left to right through a document in the way execution would in a conventional programming language. TeX expands tokens lazily, one at a time, only as they are needed by whatever process is currently consuming input, whether that is the paragraph builder, a conditional, or another macro's argument-gathering.

This produces two behaviors that are easy to misjudge without the model in hand. First, an argument to a macro is not evaluated before the macro is called; it is handed over as an unexpanded token sequence, and whether or how it gets expanded depends entirely on what the macro's definition does with it. A macro that never uses its argument, or uses it inside a context that suppresses expansion such as a section title written to an auxiliary file, may leave nested macros in that argument completely unexpanded, which is why commands like `\protect` and the `\text`-style variants for headings and captions exist at all: they manage exactly this asymmetry between where a token sequence appears and where it is ultimately consumed.

Second, controlling when expansion happens, rather than only what a macro expands to, is itself a design tool. The primitive `\expandafter` lets a macro writer reach past one token to expand the one behind it first, which is the standard technique for building a new control sequence name out of pieces, or for testing the first character of an argument before deciding how to process the rest. Packages built for robust macro programming, such as `expl3`, replace this style of manual token-shuffling with a more disciplined function-based interface, but the underlying model they sit on top of is the same lazy, rule-based expansion described here.

**Example.** Consider a macro intended to typeset a term and, the first time it is used, also add that term to a glossary:

```
\newcommand{\term}[1]{\textit{#1}\glossaryadd{#1}}
```

Here `#1` is not a value computed once and reused; it is a token sequence substituted verbatim into the macro's replacement text at every point it appears. If the argument itself contains a macro, such as a cross-reference or another custom command, that inner macro will be expanded independently, in place, whenever and wherever the surrounding context causes expansion to reach it, not once at the moment `\term` is written.

None of this is presented for its own sake. The chapters that follow, on writing key-value interfaces, designing document classes, and building large multi-file projects, all depend on treating macros as rewrite rules operating on token streams rather than as function calls in a conventional sense. A superuser's fluency with LaTeX comes less from memorizing a larger command vocabulary than from being able to predict, from the model in this chapter, what a given piece of source will actually become once TeX has finished reading it.

# Chapter 2

# Why LaTeX Works the Way It Does

TeX and LaTeX are often treated as synonyms, but the distinction between them is the key to understanding why the system behaves the way it does, including the parts that feel inconsistent until the distinction is made explicit. TeX is Donald Knuth's typesetting engine [1]: a fixed, extremely stable set of primitives for boxes, glue, penalties, fonts, and macro expansion, essentially unchanged since the 1980s. LaTeX is a large body of macros, written in TeX's own macro language, that sits on top of those primitives and gives them names and conventions a document author can use without engaging the primitives directly. `\section`, `\begin{itemize}`, and `\cite` are not part of TeX; they are LaTeX macros that eventually expand down into TeX primitives such as box construction, penalty insertion, and font selection.

## 2.1 Content and Convention

This layering was a deliberate design choice, not an accident of history. Knuth built TeX to be a stable, low-level typesetting engine and explicitly left higher-level document conventions to be built separately, on top of it, by whoever needed them. Leslie Lamport's LaTeX [3] was one such layer, built to let an author write in terms of document structure (sections, theorems, references, bibliographies) rather than in terms of layout primitives (boxes, glue, explicit spacing). The practical consequence is that a LaTeX command is almost never a single behavior; it is a name for a bundle of lower-level TeX operations that LaTeX's author decided should happen together.

This is why two different LaTeX installations, or two different versions of a class file, can render the same source differently even though neither one is "wrong": the primitives TeX exposes are stable, but the conventions LaTeX builds on top of them are a design decision, open to revision, and frequently revised. A superuser benefits from being able to look past the convention to the primitive underneath it when a convention does not do what is needed, rather than searching for a package that happens to already provide the exact combination of primitives required.

## 2.2 Macro Expansion Is Rewriting, Not Procedure Calling

It is tempting, especially for anyone coming from a conventional programming background, to read a LaTeX document as a sequence of procedure calls: `\section{Introduction}` looks like a function invocation, and it is natural to imagine TeX "running" the document from top to bottom the way an interpreter runs a script. The actual model, introduced in the previous chapter and worth restating here in this context, is closer to term rewriting. A macro is a rule that says: wherever this pattern of tokens appears, replace it with that sequence of tokens, and keep rewriting until no more expandable tokens remain in the position currently being processed.

The practical effect of this distinction shows up constantly in intermediate LaTeX use. A macro cannot, in general, "return early" the way a function can, because there is no call stack to return from, only a token stream being progressively rewritten. Recursion in TeX macros is not a language feature bolted on top of an imperative core; it falls directly out of the rewriting model, since a macro is permitted to expand to a token sequence that includes its own name, and expansion simply continues until some conditional stops it. Conditionals themselves (`\ifnum`, `\ifx`, and the LaTeX3 boolean and predicate machinery built on top of them) are not control-flow statements in the procedural sense; they are themselves expandable, producing one branch's tokens or the other's, which then continue to be rewritten like anything else.

Treating macros as rewrite rules rather than procedures also explains a class of bugs that otherwise seem inexplicable: a macro whose definition looks correct but produces the wrong output because it was invoked in a context where full expansion had already happened, or had not yet happened, by the time TeX needed a fully expanded value. Diagnosing this kind of failure is a matter of tracing where in the rewriting process a given token sequence currently sits, not a matter of stepping through a call stack.

## 2.3 Where LaTeX Breaks the Abstraction

No layer built on top of a lower one hides that lower layer perfectly, and LaTeX is no exception. There are places where the document-structure abstraction LaTeX offers leaks, and a superuser needs to recognize these leaks rather than be surprised by them.

Fragile commands are the clearest example. Certain LaTeX constructs, particularly those involving arguments that get written to an auxiliary file or reprocessed in a different context (section titles going into the table of contents, captions going into a list of figures), are not fully expanded in the context where they are written; they are expanded again later, in a different context, when the auxiliary file is read back in. A command that behaves correctly the first time but breaks when it is reprocessed this way is described as fragile, and `\protect` exists specifically to suppress expansion at the first pass so the correct behavior happens at the second. This is not a design flaw so much as a visible seam between the token-rewriting substrate and the document-structure convention layer built on top of it.

Package interaction is a second seam. Because LaTeX macros can redefine other macros, and because the order in which packages are loaded determines which definition wins when two packages try to modify the same behavior, the outcome of loading a set of packages is not simply the union of what each package does in isolation; it depends on load order and

on which package's redefinition happens to be the one still in effect when the document is typeset. This is a direct consequence of building convention out of the same rewriting mechanism an author's own macros use: packages are not sandboxed from each other or from the document.

A third seam is engine dependence. Primitives available in one engine (`\pdfximage` in classic pdfTeX, direct OpenType font loading in LuaLaTeX and XeLaTeX) are not available in others, so a document that reaches past LaTeX's abstraction layer to use an engine-specific primitive stops being engine-agnostic at exactly that point, even though the surrounding LaTeX conventions remain the same.

None of these seams are reasons to avoid working close to the primitive layer. They are, if anything, the reason to understand it: a superuser who knows where LaTeX's abstraction is thin can predict which constructs will misbehave under reprocessing, package interaction, or engine change, and can either avoid them or work around them deliberately, rather than discovering the seam by accident in a five-hundred-page document three days before a deadline.

# Part II

# Architecture

# Chapter 3

# Document Architecture, Not Document Syntax

Most LaTeX instruction stops at the level of syntax: how to write a section heading, how to insert a figure, how to format a table. That level of instruction is necessary but not sufficient for anyone working on a document longer than a handful of pages, because syntax says nothing about how to organize a project once it grows past the point where a single file is still convenient to write, review, and rebuild. This chapter is about the decisions that come before any of that syntax is written: how a book-length project is laid out on disk, how its parts relate to one another, and how that structure is kept consistent as the project grows over months or years rather than days.

## 3.1 Designing a Project Before Writing a Sentence

A short paper can be a single `.tex` file without cost. A book cannot, and the reasons are practical rather than aesthetic. A single-file monograph becomes slow to compile, since every edit forces the entire document to be reprocessed; it becomes difficult to review, since a diff against the previous version mixes changes from unrelated chapters together; and it becomes difficult to reason about, since finding the source of a given passage means searching rather than navigating. Treating chapters, appendices, figures, and the bibliography as separate concerns from the outset avoids all three problems, and doing so before the first chapter is drafted is considerably easier than retrofitting structure onto a project that has already grown large as a single file.

The separation that matters most is between content and assembly. Each chapter should live in its own file, containing only that chapter's prose, definitions, and figures; a single top-level file should contain the document class, the preamble, and a sequence of `\input` or `\include` commands that assembles the chapters into the finished book. This means a chapter file is never compiled in isolation without the correct preamble, but it also means a chapter can be edited, reviewed, and reasoned about as a self-contained unit, and that the top-level file serves as a table of contents for the source itself, not only for the rendered output.

## 3.2 `\input` versus `\include`

LaTeX offers two commands for pulling one file into another, and the difference between them matters for a project of any real size. `\input` performs a direct textual inclusion: the named file's contents are read in place, with no side effects beyond whatever the file itself does. `\include` does more: it forces a page break before and after the included file, and, critically, it interacts with `\includeonly` to allow partial compilation, where only the listed files are actually typeset while the others are skipped, with their page numbers and cross-references still available from the existing auxiliary files.

For a book-length project under active revision, this distinction is not a minor convenience. Recompiling a five-hundred-page manuscript to check a single paragraph's line breaks is slow enough to disrupt a working rhythm; recompiling only the chapter currently being edited, while still resolving references into the rest of the book correctly, is not. The tradeoff is that `\include` forces page breaks between included files, which is appropriate at chapter boundaries but wrong for finer-grained splitting, such as separating a chapter's figures or a long proof into their own file purely for editing convenience. A common and effective convention is to use `\include` at the chapter level and `\input` for everything finer, so that partial compilation aligns with chapter boundaries while sub-chapter organization stays flexible.

## 3.3 Directory Conventions for a Large Project

Once a project spans dozens of chapter files, generated figures, a bibliography, and assorted build artifacts, an unstructured flat directory becomes as much of an obstacle as an unstructured single file was. A directory layout that separates these categories from the start pays for itself repeatedly: a `chapters/` directory holding one file per chapter, numbered or named so their order is legible without opening them; a `figures/` directory holding source images and, separately, a location for figures generated by an external script or program, so that regenerated output does not get confused with hand-authored artwork; a `bib/` directory holding the bibliography database or the `thebibliography` source, kept as its own file even when the project is not using BibTeX, so that citation data is not duplicated across chapters; and a build directory, excluded from version control, holding the `.aux`, `.log`, `.toc`, and other files a compiler regenerates on every run.

This layout does more than keep the project tidy. It makes the boundary between what is authored and what is generated explicit, which matters once a project starts pulling figures from simulation output or data pipelines, a workflow covered later in this book. A generated figure that lives in a directory clearly marked as generated can be deleted and rebuilt without anxiety; a generated figure indistinguishable from a hand-drawn one cannot, and the distinction is easy to lose once a project has been worked on intermittently for a year or more.

## 3.4 Consistency Across Files

Splitting a document into many files introduces a problem a single file never has: keeping definitions, notation, and terminology consistent across pieces that are edited separately,

sometimes months apart. A macro defined only in the preamble of the top-level file is available everywhere, which is the right home for anything used across more than one chapter; a macro defined inside a chapter file and needed elsewhere is a sign that it belongs in a shared preamble file instead, included once and referenced by every chapter that needs it.

Cross-references are the sharpest version of this problem. A `\ref` or `\cite` in one chapter file depends on a `\label` or bibliography entry that may live in a different file entirely, and the two are connected only through the auxiliary files LaTeX writes during compilation, not through anything visible in the source itself. This is why a full project typically needs two compilation passes at minimum: the first pass records every label's position and every citation's use, and the second resolves references against that record. A `\ref` that renders as a question mark, or a citation that renders as `[?]`, almost always means either that a second pass has not yet run or that the file defining that label or entry has not been compiled at all, which happens easily in a multi-file project if `\includeonly` is left restricting compilation to a subset of chapters while a reference to another chapter is added.

The underlying requirement, stripped of LaTeX-specific mechanics, is that the pieces of a large document have to agree with each other wherever they touch: a term used in Chapter 12 has to match its definition in Chapter 3, a label referenced in the introduction has to exist somewhere in a later chapter, and the bibliography has to contain every work any chapter cites. None of this is enforced automatically; it is enforced by the discipline of the directory and file conventions in this chapter, and by treating a failed cross-reference or an inconsistent term not as a typo to patch locally but as a sign that two pieces of the document have drifted out of agreement and need to be reconciled.

The remainder of Part II builds on this foundation: version control conventions for a project organized this way, and the build tooling that makes recompiling a project of this size fast enough to work with on a daily basis.

# Chapter 4

# Version Control for Publishing Systems

A book-length LaTeX project under active development for months or years is, in every respect that matters for tooling, a software project. It has source files that are edited incrementally, a build process that turns those sources into a deliverable, artifacts that should not be confused with source, and a history worth preserving so that earlier states of the work remain recoverable. Git, or any version control system, is the natural tool for managing that history, but applying it well to a publishing project requires a few decisions that do not arise, or arise differently, in software development.

## 4.1 What Belongs in the Repository

The chapter files, the preamble, the bibliography source, hand-authored figures, and the top-level assembly file are unambiguously source: they are what a human wrote, and nothing else in the project can reconstruct them if they are lost. These belong in version control without qualification.

Everything LaTeX generates during compilation belongs on the other side of the line. The `.aux`, `.log`, `.toc`, `.lof`, `.lot`, `.out`, and `.fls` files, along with the compiled `.pdf` itself in most workflows, are fully reproducible from the source files given the same engine and package versions, and tracking them in version control adds noise without adding information: every compilation changes them, so every commit that touches the document also appears to touch dozens of files that carry no authored content. A `.gitignore` file listing these extensions, set up once at the start of a project, keeps the repository's history focused on what was actually written rather than on what was regenerated.

Generated figures sit in a middle position that deserves a deliberate choice rather than a default. A plot produced by a script from a data file is, like a compiled `.aux` file, reproducible from its inputs, and an argument can be made for excluding it from version control the same way. In practice, most publishing projects are better served by tracking generated figures anyway, for a reason that has nothing to do with reproducibility in principle and everything to do with reproducibility in practice: the script, the data, and the exact library versions that produced a given figure six months ago may not all still be available or working today,

while the figure itself, once generated, is stable. Treating generated figures as source once they are committed, while keeping the generating script alongside them so the figure can be regenerated when needed, is usually the more robust choice for a project meant to remain buildable years later.

## 4.2 Structuring Commits Around Content, Not Around Builds

A commit history is only useful if it can be read later, and a history dominated by commits like "fix typo," "recompile," and "update," with no indication of which chapter or which idea each one touches, is not readable in that sense even though it is technically complete. The discipline worth adopting is committing around units of authored content: a commit that finishes drafting a section, one that revises a chapter's argument in response to feedback, one that adds a new theorem and its proof. This is a matter of message quality more than of commit frequency; small, frequent commits are fine, even desirable, as long as each one has a message that describes what changed at the level of the document's ideas rather than at the level of "saved file."

Because chapters live in separate files, as established in the previous chapter, this discipline is easier to maintain than it would be in a single monolithic document: a commit that only touches `chapters/12-bibliography.tex` is self-evidently about that chapter's bibliography material, and a diff against the previous version of that file shows exactly what changed in that chapter's argument, uncomplicated by unrelated edits elsewhere in the book. This is one of the practical payoffs of the file-per-chapter architecture from Chapter 3, beyond the compilation-speed benefits already discussed there.

Tagging is worth using deliberately in a publishing project, distinct from its typical use in software for marking releases. A tag placed at the point a manuscript is sent for review, submitted to a publisher, or finalized for a particular edition gives a permanent, named anchor to return to, independent of whatever branch or commit history accumulates afterward. For a project that produces multiple related books or revised editions over time, as a long-running research program typically does, these tags become the record of what each published version actually contained, which is otherwise easy to lose once further revision has moved the working copy well past what any given reader actually saw.

## 4.3 Reproducibility as a Discipline

Reproducibility in a publishing context means more than "the source is in Git." It means that a checkout from any point in the project's history can be rebuilt into the same PDF it produced at that time, by anyone, not only by the original author on the original machine. This depends on more than the `.tex` files: it depends on which packages were available, which versions of those packages, which font files were installed, and which engine was used.

Most of this dependency information is invisible unless it is made explicit. A short note in the repository, listing the intended engine (LuaLaTeX, in the conventions used throughout this book), the minimum LaTeX and package versions the document relies on, and any non-standard fonts the document requires along with where to obtain them, costs little to maintain and is the difference between a project that can be picked up again

after a year of inactivity and one that requires reconstructing a forgotten environment by trial and error. For projects using package managers such as TeX Live's own version pinning or containerized build environments, recording the exact environment mechanically is preferable to a hand-maintained note, but even a hand-maintained note is far better than nothing, and costs a fraction of the effort.

The build directory convention introduced in the previous chapter supports this discipline directly: because nothing generated is tracked, a clean checkout followed by a single build command is the only way to produce a working PDF, which means that command is exercised every time the project is picked up again, rather than being allowed to silently rot while a stale compiled PDF sits untracked on a single machine and is mistaken for a reliable record of the source's current state.

# Chapter 5

# Build Systems and Continuous Compilation

The previous two chapters established how a large project's source should be organized and tracked. This chapter is about the step that turns that source into a finished PDF: running the compiler enough times, in the right order, with the right auxiliary tools, to produce correct output, and doing so with as little manual intervention as the project's size demands. A single-page document can be built by typing `lualatex file.tex` and glancing at the result. A book with a bibliography, an index, and cross-references scattered across forty chapter files cannot, not because any individual step is hard, but because the sequence of steps has to be run correctly and in order every time, and a human doing that by hand will eventually skip a step, run it out of order, or forget it entirely under deadline pressure.

## 5.1 The Multi-Pass Problem

LaTeX's compiler processes a document from beginning to end in a single pass, but several features a book-length document depends on cannot be resolved in a single pass in principle, not merely as an implementation limitation. A table of contents needs to know every section's title and page number before it can be typeset, but the table of contents itself typically appears near the beginning of the document, before most of those sections have been processed. A cross-reference to "see Chapter 12" needs to know what page Chapter 12 ends up on, which is not settled until the entire document, including everything after the reference, has been laid out.

LaTeX resolves this by writing what it learns during one pass into an auxiliary file, the `.aux` file, and reading that file back in at the start of the next pass. On the first pass, every `\label` records its current page and section number into the `.aux` file, but every `\ref` to a label defined later in the document has nothing to read yet, since that label has not been processed and its position has not been recorded, so it renders as a placeholder question mark. On the second pass, the `.aux` file now contains every label's position from the first pass, so every reference resolves correctly, assuming nothing in the document has shifted enough between passes to invalidate what the first pass recorded. In practice, a document with many cross-references sometimes needs a third pass, since a reference resolving on the

second pass can itself shift page breaks enough to move a label from the first pass, and only stabilizes once a full pass makes no further changes.

Bibliographies and glossaries follow the same pattern with an extra step: BibTeX or Biber reads the citations recorded by a first LaTeX pass, produces a formatted bibliography, and that bibliography then needs to be pulled back into the document on a subsequent LaTeX pass, meaning a document with a bibliography typically needs LaTeX, then the bibliography tool, then LaTeX again at minimum, and often once more to let cross-references to bibliography entries settle. An index built with `makeindex` and a glossary built with the `glossaries` package's own processing tool add further instances of the same pattern: something outside LaTeX itself has to run between two LaTeX passes to produce a resource the second pass consumes.

## 5.2 Automating the Sequence

None of this sequence is complicated to describe, but a sequence of five or six commands, run in the correct order, with the correct arguments, every time any file in the project changes, is exactly the kind of task that should not be left to memory. LATEXMK exists to solve this directly: given a top-level `.tex` file, it inspects what auxiliary tools the document actually needs (by watching for `.bib`, `.idx`, or glossary files it needs to process), runs the correct sequence of compiler and auxiliary-tool invocations automatically, and, critically, keeps rerunning until the output stabilizes rather than running a fixed number of passes regardless of whether they were sufficient. A project set up with a `latexmkrc` file specifying the engine (LuaLaTeX, per this book's conventions) and any non-default options needs nothing more than LATEXMK typed at the top level to produce a correct, fully resolved PDF, regardless of how many auxiliary tools the current draft happens to require.

ARARA takes a different approach to the same problem, driven by directives written as comments inside the `.tex` file itself, specifying which tools to run for that particular document. This is useful when different documents in the same project need different build sequences, or when the build sequence needs to be readable by someone opening the source file itself rather than a separate configuration file. For a single coherent book project with one consistent build sequence throughout, LATEXMK's external configuration is usually the better fit; for a collection of loosely related documents with varying needs, ARARA's per-file directives can be the more maintainable choice.

A Makefile is worth adding on top of either tool once a project has multiple build targets: a full book, an excerpt for review, a version with tracked-changes markup visible, a print-ready PDF with different margins than the screen version. Encoding these as named Make targets, each invoking LATEXMK or ARARA with the appropriate options, turns "how do I build the review copy again" into `make review`, which is a small convenience for a single author working alone and a considerable one for a project anyone else needs to build without first reconstructing the author's usual sequence of commands from memory.

## 5.3 Continuous Compilation During Drafting

The tooling described so far handles producing a final, fully resolved PDF, but a full multi-pass build with bibliography and glossary processing is more than is needed, and often too slow to be pleasant, while actively drafting a single chapter. LATEXMK's `-pvc` (preview continuously) mode addresses this directly: it watches the project's files for changes and triggers a rebuild automatically on save, paired with a PDF viewer that reloads the output, so that the feedback loop between writing a sentence and seeing it typeset shrinks to the time a single build takes rather than requiring a manual command after every edit.

This matters more for a project organized the way Chapter 3 describes than it would for a single-file document, because a chapter-scoped edit only needs to rebuild quickly if the build tooling is actually taking advantage of that scoping. Combining `\includeonly` with a continuous build restricted to the chapter currently being drafted keeps the feedback loop fast even in a project whose full build, bibliography and all, takes long enough to be disruptive if triggered on every keystroke-adjacent save. The full build remains necessary before anything is considered finished, since a chapter-scoped build will not catch a broken cross-reference into a chapter that was excluded, but it does not need to run on every edit, only at the points where the document actually needs to be checked as a whole.

With the project organized, tracked, and buildable on demand, the next part of the book turns to the layer above all of this infrastructure: the macros and interfaces an author actually writes with, and how to design them so that a project's growing complexity is managed by the tooling rather than by the author's memory.

# Part III

# Programming in LaTeX

# Chapter 6

# Macros as Terms

Part I established that TeX processes a document by rewriting tokens rather than executing procedures, and Part II covered how to organize the files a document is built from. This chapter turns to the material an author actually writes when extending LaTeX: macros. Understanding macros correctly, as the rewrite rules described in Chapter 1 rather than as function definitions in a conventional programming language, is what separates writing macros that work by luck from writing macros whose behavior can be predicted before they are ever run.

## 6.1 `\def` and `\newcommand`

TeX itself provides exactly one primitive for defining a macro: `\def`, which takes a control sequence name, an optional parameter text describing how arguments are delimited, and a replacement text. LaTeX's `\newcommand` and `\renewcommand` are built on top of `\def`, adding a small amount of safety: `\newcommand` checks that the name being defined is not already in use and raises an error if it is, `\renewcommand` checks the opposite, and both provide a more convenient syntax for the common case of a macro taking a fixed number of simple, undelimited arguments, `#1` through `#9`.

The relationship between the two matters because `\def` remains available and is sometimes the better tool. `\newcommand` cannot redefine an existing name without `\renewcommand`, cannot easily define a macro with a delimited argument (one that reads up to a specific token rather than a fixed number of braces), and adds a small amount of overhead in exchange for its safety checks. A macro author working inside LaTeX's conventions reaches for `\newcommand` by default, since the safety it provides catches a real class of accidental redefinition errors, but recognizing when `\def`'s lower-level flexibility is actually needed, rather than working around `\newcommand`'s limitations with increasingly contorted workarounds, is part of what distinguishes fluent macro writing from macro writing that happens to work.

## 6.2 Expansion versus Execution

Chapter 1 introduced the distinction between tokenization and expansion; macro writing is where that distinction has direct, daily consequences. Some TeX operations are expandable:

they can occur inside contexts that require full expansion before anything else happens, such as the argument to `\edef` or a conditional's test, and their job is to produce more tokens for further processing. Other operations are not expandable at all; they have side effects, such as changing a register's value or starting a new paragraph, and cannot meaningfully occur in a context that only wants tokens.

A macro defined with `\def` or `\newcommand` is, by default, expandable, in the sense that its name gets replaced by its body wherever expansion reaches it. But the body of that macro is free to contain non-expandable material, in which case the macro as a whole behaves as a mix: expanding partially to reveal its non-expandable content, at which point expansion of that particular token stops, since there is nothing further to expand at that position. This is why some macros can be used inside a context like a section title that gets written to an auxiliary file, and others cannot: the ones that can are built entirely from expandable material all the way down; the ones that cannot contain something, even indirectly, that only makes sense to execute rather than expand.

`\edef`, as opposed to `\def`, defines a macro whose body is fully expanded once, at definition time, rather than left to be expanded later wherever the macro is used. This is the standard tool for "freezing" the current value of something that would otherwise change, such as capturing the current value of a counter into a fixed string rather than a reference that would track the counter's later changes. Confusing `\def` and `\edef` is one of the most common sources of a macro that behaves correctly the first time it is used and incorrectly the second, because `\def` captured a reference to something mutable while `\edef` would have captured its value at that moment.

## 6.3 Argument Specification and `xparse`

A macro defined with plain `\newcommand` takes a fixed number of simple arguments, each either mandatory or, for at most one argument, optional with a default value. Many of the macros an author actually wants to write need more than this: an argument that is optional but has no sensible single default, an argument delimited by a specific character rather than braces, or a variable number of arguments depending on what was provided. The `xparse` package, part of the LaTeX3 kernel and available for use regardless of whether the rest of a document uses LaTeX3 syntax, provides a compact notation for specifying exactly this kind of argument structure: `m` for a mandatory argument, `o` for optional, `s` for an optional star, `d` for a delimited argument with custom delimiters, among others, combined into a single specifier string that describes a macro's full calling convention in one place.

This specifier string is worth thinking of as a lightweight signature for the macro: it says, in a compact and checkable form, what a call to this macro is required to look like and what forms are permitted, independent of what the macro's body actually does with the arguments once they are captured. A macro defined with `\NewDocumentCommand{\keyword}{m o}{...}` states plainly, at the point of definition, that a call takes one mandatory argument and one optional one, which is more legible than inferring the same fact from reading a `\def`-based implementation that manually checks for and strips an optional square-bracket argument by hand. For a macro whose calling convention will be used by anyone other than the person who wrote it, including a future version of the same author, that legibility is worth the small overhead of loading `xparse`.

## 6.4 Writing Macros That Predict Their Own Behavior

The point of treating macros as terms rather than procedures is not merely conceptual accuracy for its own sake; it changes how a macro should be designed. A macro intended for use inside contexts requiring full expansion (section titles, index entries, anywhere text is written to an auxiliary file and reread) has to be built from fully expandable pieces throughout its call chain, not merely at its own top level, since a macro that itself expands cleanly but calls another macro internally that does not will inherit that inner macro's limitation. Checking this by working through what a macro expands to, step by step, in the specific context it will actually be used, catches this class of error before it surfaces as a mysterious failure three chapters and several months later, when the macro is finally used somewhere its original author never tested.

The chapters that follow build directly on this foundation: key-value interfaces, covered next, are a more elaborate argument-handling pattern built from the same primitives introduced here, and robust command design, closing out this part of the book, is largely the discipline of applying the expansion-versus-execution distinction consistently across every macro a project depends on.

# Chapter 7

# Conditionals, Loops, and Key-Value Interfaces

A macro that always does the same thing with a fixed set of arguments is only useful up to a point. Most of the macros a superuser ends up writing for a large project need to behave differently depending on what they are called with, or need to process a variable amount of input, or need to expose a set of independent options an author can turn on or off without the macro's calling convention becoming an unreadable string of positional arguments. This chapter covers the tools for all three: conditionals, loops, and key-value interfaces, the last of which is the pattern most worth learning well, since it is what makes a package's public interface usable by someone other than its author.

## 7.1 Conditionals

TeX's native conditionals, `\ifnum`, `\ifdim`, `\ifx`, and the small family of related primitives, are expandable, in the sense introduced in the previous chapter: they can appear inside a context requiring full expansion, and they produce one branch's tokens or the other's rather than executing a statement. `\ifx` in particular is worth understanding well beyond its common use for comparing two macros for equality, since it compares the current meaning of two tokens, which makes it the standard tool for testing whether a macro has already been defined, whether an optional argument was left empty, or whether two pieces of user input match, all without needing anything beyond TeX's own primitives.

The `etoolbox` package builds a more approachable layer on top of these primitives: `\ifdefined`, `\ifboolexpr`, and a family of list- and string-testing conditionals that read closer to their intent than the equivalent primitive expression would. For an author who does not need to reach past this layer, `etoolbox`'s conditionals are usually the better choice on readability grounds alone; understanding what they expand to in terms of the primitives from the previous section remains valuable for the cases where `etoolbox`'s conditional does not quite cover what is needed and a primitive-level conditional has to be written by hand.

## 7.2 Loops

TeX has no native looping construct beyond the recursive self-reference a macro can make to its own name, described in Chapter 2 as a direct consequence of the rewriting model rather than a separately implemented feature. `etoolbox` provides `\forcsvlist` and related commands for iterating over a comma-separated list, which covers the overwhelming majority of looping an author actually needs in document preparation: applying the same formatting to each item in a list of keywords, processing each file in a list of includes, or repeating a pattern once per entry in a small, explicitly given set. For anything resembling a numeric loop, incrementing a counter and repeating a body a fixed number of times, `\prg_replicate:nn` from the LaTeX3 kernel, or the older `\@whilenum` from the LaTeX2e kernel, provide the same capability at different levels of the toolchain, with the LaTeX3 version generally preferable in new code for its more predictable expansion behavior.

The practical guidance worth taking from this section is restraint: a loop that runs once per chapter to build a table of contents entry, or once per item in a short, fixed list, is a reasonable and common thing to write. A macro-level loop standing in for what would be a straightforward script-level loop over external data, such as generating a hundred entries from a spreadsheet, is usually better handled by generating the LaTeX source for those hundred entries with an external script and including the result, rather than by pushing that iteration into TeX's comparatively awkward looping tools. Knowing where that line sits, between what belongs inside the document's own macros and what belongs in a preprocessing step outside LaTeX entirely, is itself part of a superuser's judgment.

## 7.3 Key-Value Interfaces

A macro with more than two or three optional behaviors quickly becomes unusable if each behavior is controlled by its own positional argument, since the calling code has to remember what each position means and provide placeholder values for anything left at its default. Key-value syntax solves this by letting each option be named explicitly at the call site: `\mycommand[color=blue, bold=true]{text}` states directly what each setting controls, in any order, with unneeded options simply omitted rather than filled in with a placeholder.

`pgfkeys`, originally built for PGF and TikZ but usable independently, and `l3keys`, the LaTeX3 kernel's own key-value system, both provide this pattern as a general tool rather than something tied to graphics. Both let a macro author define a tree of keys, each with a default value, a handler describing what happens when the key is set, and, in `l3keys`, an optional type check on the value provided. Choosing between them is mostly a matter of context: a project already using TikZ heavily has `pgfkeys` loaded regardless, while a project building on the LaTeX3 kernel throughout benefits from `l3keys`'s tighter integration with the rest of that toolchain.

Designing a key-value interface well is closer to designing a small API than to writing an ordinary macro, and the same considerations apply. Every key should have a sensible default, so that a call providing none of the optional keys still produces reasonable output; keys should validate their input where practical, so that a typo in a value produces an immediate, legible error rather than a confusing failure several steps removed from its cause; and keys should compose, meaning that setting one key should not silently change

the meaning of another unless that interaction is the documented, intended behavior. A key-value interface that violates any of these is usually harder to use than the positional-argument macro it replaced, since it adds the overhead of remembering key names without delivering the benefit of predictable, independent behavior that was the point of moving to keys in the first place.

## 7.4 An Example Worth Working Through

Consider a macro for typesetting a labeled example box, the kind that recurs throughout a book like this one, that should support an optional title, an optional numbering scheme independent of the book's other counters, and an optional flag for whether the box should be boxed with a visible border or left as plain indented text. Written with positional arguments, this macro's calling convention would depend on argument order and would need placeholder values at every call site that wanted to set the third option without also setting the first two. Written with `l3keys`, the same macro exposes `title`, `numbered`, and `boxed` as independently settable keys, each with its own default, callable as `\examplebox[boxed=false]{...}` without needing to know or provide anything about the other two options. The interface, not merely the implementation behind it, is what makes the macro pleasant to use across a five-hundred-page book where it might be called differently in dozens of places.

The next chapter turns from what a macro's interface looks like to how its internals should be written so that it behaves correctly and predictably regardless of the surrounding context it is used in, closing out Part III before the book moves on to typography and fonts.

# Chapter 8

# Robust Command Design

The preceding chapters covered how macros are defined, how expansion behaves, and how to give a macro a clean calling interface. This chapter is about a different concern: making sure a macro keeps behaving correctly once it leaves the context it was first tested in, gets used by someone other than its author, or has to coexist with dozens of other macros in a project that has been growing for years. None of the individual techniques here are difficult; the discipline is in applying them consistently across an entire project rather than only where a bug has already shown up.

## 8.1 Fragile Commands, Still

Chapter 2 introduced fragile commands as one of the places LaTeX's abstraction visibly leaks: a macro that works correctly the first time it is used but breaks when its argument is written to an auxiliary file and reprocessed, because that reprocessing happens in a context where full expansion is required and the macro was not built to survive it. This might seem like a legacy concern from an older, pre-LaTeX3 era, but it has not gone away in modern engines. Section titles, captions, and running headers are still written to auxiliary files and reread; a macro used inside any of these that is not built from fully expandable material, all the way down through anything it calls internally, will still fail in exactly the way it failed decades ago, regardless of whether the surrounding document uses LuaLaTeX, XeLaTeX, or the classic pdfTeX engine.

The practical response has two parts. The first is `\protect`, which prevents a fragile command from being expanded prematurely in a moving-argument context, deferring its expansion to the point where it is actually being typeset rather than the point where it is being recorded into an auxiliary file. The second, and generally the better long-term fix where it is available, is writing the macro to be robust in the first place, built entirely from expandable primitives or from `\newcommand`, `\NewDocumentCommand`, and other LaTeX-kernel definition commands, which produce robust macros by default in current LaTeX releases. `\protect` remains necessary for older or lower-level macro definitions and for third-party commands whose internals are not under the author's control; for anything written from scratch for a new project, preferring robust-by-construction definition commands avoids the problem before it starts.

## 8.2 Namespacing

A large project accumulates macros quickly: helper commands for notation, shortcuts for repeated phrases, formatting wrappers around theorem environments, and macros supporting the book's own custom document class. Left unnamespaced, this accumulation eventually collides, either with a macro name the author reused without noticing, or with a name a loaded package happens to define for its own purposes, silently overriding one definition with the other and producing a failure that surfaces far from its actual cause.

The LaTeX3 convention of `\ClassName_module_function:nn`-style naming, using underscores and colons reserved for package and class code rather than document-level macros, is one solution, but it is heavier than most single-author book projects need throughout. A lighter convention that captures most of the benefit is prefixing project-specific macros with a short, consistent tag tied to the project or the book's own notation system: `\rsvpfield`, `\rsvpboundary`, rather than `\field` and `\boundary`, which are far more likely to collide with something a loaded package already defines. This costs a small amount of typing and pays for itself the first time a package update introduces a name that would otherwise have silently shadowed a document-level macro with the same name, a failure mode that is often diagnosed by symptom rather than by cause unless the naming convention is disciplined enough to make the collision itself unlikely.

For a project spanning multiple related books sharing a common notation system, this namespacing matters even more, since macros defined for one book's specific needs are liable to be copied into a companion project's preamble, and an unprefixed macro name is far more likely to already mean something different in the second project than a prefixed one is.

## 8.3 Writing for a Future Reader

The macros in a large project are read far more often than they are written: read while debugging, read while extending, read by a collaborator, read by the same author eighteen months later with no memory of the reasoning behind a particular implementation choice. A macro's definition should be legible on its own terms, which in practice means a small set of habits applied consistently. A comment above any macro whose purpose is not obvious from its name and arguments, stating what it is for rather than restating what the code already shows syntactically. Meaningful argument names in `xparse` and `l3keys` definitions where the tooling allows it, rather than relying on positional `#1`, `#2` with no accompanying explanation of what each position means. Consistent formatting of macro definitions across a project, so that a reader's eye can find the argument list, the body, and any embedded conditionals in the same relative position every time, rather than having to re-parse an unfamiliar layout for every new macro encountered.

None of this is about elegance for its own sake. A project that will be actively maintained, extended, and rebuilt for years benefits from exactly the same discipline that a long-lived software codebase benefits from, and for the same reason: the cost of an illegible macro is paid not once, at the moment it is written, but repeatedly, every time it has to be understood again by someone who is not currently holding its full context in mind.

Part III closes here, with the tools for writing macros that behave predictably and

interfaces that are pleasant to call. Part IV turns to typography and fonts: the layer where LuaLaTeX and XeLaTeX's modern capabilities are put to direct use, after three chapters spent entirely on how a document is built and programmed rather than on how it looks.

# Part IV

# Typography and Fonts

# Chapter 9

# Modern Font Technology with LuaLaTeX and XeLaTeX

For most of LaTeX's history, using a font meant using one of a small set of Type 1 PostScript fonts specially prepared for TeX, each requiring its own metric files and encoding support before it could be used at all. This is still how the original pdfTeX engine works, and it is why pdfTeX-based documents are limited to a comparatively small library of fonts unless someone has already done the preparation work. LuaLaTeX and XeLaTeX exist in part to remove this limitation: both can load ordinary OpenType and TrueType font files directly, the same font files used by any other modern application, with no separate preparation step required. This chapter covers what that capability actually makes possible and where the two engines still differ enough to matter.

## 9.1 Why Not pdfLaTeX

pdfTeX remains the fastest of the three engines and is still the right choice for documents that only need Computer Modern or one of the small set of professionally prepared Type 1 fonts already packaged for TeX. For anything else, and especially for a book-length project intended to use a specific, deliberately chosen typeface, pdfTeX is the wrong default. Using an arbitrary OpenType font under pdfTeX requires generating Type 1 metric files for it first, a process that is at best an extra build step and at worst does not work cleanly for a given font at all, particularly for fonts with modern OpenType features that have no equivalent in the older metric format pdfTeX expects.

LuaLaTeX and XeLaTeX both sidestep this entirely: a font is specified by name or by file path, loaded directly by the engine, and its full feature set becomes available through fontspec without any separate preparation. For a project committed to a specific typeface as part of its visual identity, as this book's own conventions are (TeX Gyre Pagella throughout), this is not a marginal convenience; it is the difference between the font being usable at all and requiring a preparation step that may not succeed.

## 9.2 fontspec and OpenType Features

fontspec is the interface both LuaLaTeX and XeLaTeX use for font loading, and its feature syntax is worth learning in some depth, since most of what makes a document look professionally typeset rather than merely correctly typeset lives here. `\setmainfont{TeX Gyre Pagella}` loads a font by its family name; the same command accepts a bracketed list of OpenType features that changes how that font behaves well beyond simply which glyphs it contains.

Ligatures beyond the basic `fi` and `fl` pairs, historical and discretionary ligatures a font may include, are enabled or disabled through the `Ligatures` feature key. Small capitals, when a font provides a true small-caps design rather than a mechanically shrunk version of the regular capitals, are accessed through `Letters=SmallCaps` or the `\textsc` command once the feature is enabled, and the difference between a genuine small-caps design and a shrunk substitute is visible immediately to anyone who has seen both, which is part of why choosing a font family with real small capitals matters for a document that uses them at all. Old-style figures, numerals that vary in height and descend below the baseline rather than sitting uniformly at cap height, are enabled through `Numbers=OldStyle`, and suit running prose far better than the lining figures most fonts default to, which were designed to align with tables rather than to sit comfortably inside a sentence.

None of these features change what characters a document contains; they change which glyphs the font substitutes for those characters, entirely at the typesetting layer, which is precisely why fontspec's feature syntax is worth treating as part of a document's typographic design rather than as an optional flourish. A title page, front matter, and running text that consistently use a font's intended ligatures, small capitals, and old-style figures read as considered and professionally set; the same document with all of a font's default settings left untouched reads as merely competent.

## 9.3 Multiple Font Families and Mathematics

A typeset book rarely uses a single font throughout. fontspec supports loading a separate family for monospaced text (`\setmonofont`, typically used for code listings) and a separate family for sans-serif text (`\setsansfont`, used for headings in some design conventions or for figures and diagrams), each with its own independent feature set, so that a monospace font chosen for code legibility does not need to share the body text's ligature and old-style-figure settings, which would be actively wrong for code.

Mathematics is the one area where fontspec alone is not sufficient, since math typesetting needs a font specifically designed with the full set of mathematical glyphs, spacing, and sizing variants LaTeX's math mode expects, not merely a text font pressed into service. unicode-math, layered on top of fontspec, provides this, loading a dedicated OpenType math font through `\setmathfont` and giving direct Unicode input for mathematical symbols in place of the older, more limited `amsfonts`/`amssymb` symbol-by-command approach. A document committed to a particular text typeface throughout, as this book is to TeX Gyre Pagella, generally wants a math font from the same family where one exists, so that text and mathematics read as part of one coherent design rather than as two different typefaces awkwardly sharing a page; where no matching math font exists for a chosen text face,

choosing a math font that harmonizes reasonably well in weight and proportion is the next-best option.

## 9.4 Keeping a Document Engine-Agnostic

LuaLaTeX and XeLaTeX share fontspec as a common interface, which lets most font-related document code run unchanged under either engine, but the two are not identical underneath that shared interface, and a superuser working across both, or handing a document to a collaborator whose installation defaults to the other engine, needs to know where the differences actually surface.

LuaLaTeX is built on LuaTeX, which exposes a full embedded Lua scripting layer, letting a document call arbitrary Lua code during typesetting; XeLaTeX has no equivalent, so any macro relying on `\directlua` or a Lua-based package is LuaLaTeX-only by construction, not merely by convention. XeLaTeX, in turn, has historically had somewhat different and in some cases more mature support for certain complex-script shaping engines through its use of the system's native font-rendering libraries, though LuaTeX's own shaping support has closed most of this gap in recent releases. Microtype's font expansion and protrusion features, which subtly adjust glyph widths to improve justification, behave slightly differently between the two engines because they operate through different underlying mechanisms, which can produce visibly different line breaks for the same source under the two engines even with identical fontspec settings.

For a document meant to remain buildable under either engine, the practical discipline is straightforward: avoid `\directlua` and other LuaTeX-specific primitives in document-level code, even when working primarily in LuaLaTeX, unless engine-specific behavior is actually required and is documented as such; test a build under both engines at least once before considering a template or class file finished, rather than assuming shared fontspec syntax guarantees shared output; and treat any package documentation that states an engine requirement as authoritative rather than assuming a feature that works under one engine will silently work under the other. A project that follows this discipline can genuinely move between LuaLaTeX and XeLaTeX as needed; one that does not will eventually break in a way that is difficult to diagnose, because the failure will look like a font problem when its actual cause is an engine-specific primitive buried several packages deep in the document's dependency chain.

The next chapter goes further into what this font infrastructure makes possible beyond ordinary Latin-script typesetting: multilingual documents, custom notation systems, and the construction of entirely new symbol sets built on top of the same OpenType foundation covered here.

# Chapter 10

# Custom Fonts, Symbols, and Writing Systems

The previous chapter covered using an existing OpenType font well. This chapter goes further, into territory most LaTeX books never reach: loading a font that was custom-built or commissioned rather than downloaded from a standard collection, defining new mathematical notation cleanly enough that it does not collide with anything else in a document, and, for a reader who needs it, constructing an entirely new symbol system or writing system on top of the same OpenType infrastructure the rest of this book has been using for ordinary typefaces.

## 10.1 Loading and Subsetting Custom Fonts

A font a designer built specifically for a project, whether commissioned professionally or produced with a font editor such as FontForge or Glyphs, loads through `fontspec` exactly the same way a standard typeface does, provided it is a valid OpenType or TrueType file: `\setmainfont{MyCustomFace}[Path = ./fonts/]` if the font is not installed system-wide, which is the more portable choice for a project meant to be built on more than one machine, since it keeps the font file itself inside the project's own directory structure rather than depending on it being separately installed wherever the document is compiled.

Subsetting, embedding only the glyphs a document actually uses rather than a font's full glyph set, matters most for distribution rather than for compilation: a PDF with a fully embedded custom font can be large if that font contains hundreds of glyphs the document never uses, and most PDF-producing engines, LuaLaTeX included, subset embedded fonts automatically as part of PDF generation, so this is typically not something an author needs to manage by hand. Where it does become a manual concern is with fonts under a license that restricts subsetting or redistribution; a project using a commissioned or purchased font should confirm the license permits the embedding the finished PDF will contain, since font licensing terms vary considerably and are easy to overlook until a PDF is already being distributed.

## 10.2 Defining New Mathematical Notation Cleanly

Chapter 6 covered `\newcommand` and macro definition generally; mathematical notation adds a specific concern beyond ordinary macros, which is collision with symbols other packages already define. `amsmath`, `mathtools`, and any specialized package a document loads all claim command names for their own symbols, and a new operator or symbol defined without checking against what is already in use risks silently overriding an existing definition, or being silently overridden by a later package load, either of which produces a document that renders differently than intended without any error being raised.

`\DeclareMathOperator`, from `amsmath`, is the correct tool for a new named operator that should behave like `\sin` or `\lim` (upright, properly spaced from surrounding material), rather than defining it by hand with `\newcommand` and `\mathrm`, which technically produces similar-looking output but does not participate correctly in spacing decisions the way a properly declared operator does. For a genuinely new symbol rather than a named operator, `\newcommand` is appropriate, but checking `\ifdefined\symbolname` before the definition, or simply grepping the project's own macro files and its package list for the intended name before committing to it, is a small habit that prevents a class of bug that is otherwise very difficult to track down, since a shadowed symbol produces no error message at all; it simply renders as whatever definition happened to win.

For a project defining a substantial body of custom notation, collecting all of it into a single, clearly named file, loaded once from the project's shared preamble as described in Chapter 3, rather than scattering symbol definitions across whichever chapter first happened to need them, keeps the full symbol vocabulary visible in one place and makes a naming collision far easier to catch before it causes a problem.

## 10.3 Building a Private Notation System

A research program that spans many essays and books, using a recurring, specific vocabulary of symbols and operators across all of them, benefits from treating notation as something designed once and reused deliberately, rather than invented ad hoc in whichever document happens to need a new symbol first. This is a discipline decision as much as a technical one: choosing a symbol for a recurring concept, committing to it, and using it consistently across every subsequent document is what lets a reader who has encountered the notation once recognize it immediately in a different book, rather than having to relearn a locally redefined symbol every time.

Practically, this means maintaining the shared symbol file described in the previous section not per-project but across an entire corpus, included identically at the top of every book or essay that uses the notation, with new symbols added to that shared file as the framework grows rather than defined locally and only later, if ever, reconciled with the shared vocabulary. A symbol retired or renamed in the shared file should be handled the way any breaking change is handled in a maintained codebase: with a clear record of what changed and why, since older documents referencing the previous name will otherwise fail to compile, or worse, silently compile with the wrong symbol if the old name was reused for something new rather than removed outright.

## 10.4 Custom Symbols and Invented Writing Systems

For an author who needs symbols beyond what any existing font provides, whether a small set of custom glyphs for a specific notation system or something as ambitious as a complete invented script, the underlying mechanism is the same OpenType infrastructure covered in the previous chapter, used in the opposite direction: instead of loading a font someone else built, the document loads a font built specifically to contain the needed glyphs.

Building such a font is genuinely a font-design task, done with FontForge or a comparable tool, mapping each new symbol onto a Unicode code point, either from one of Unicode's officially reserved Private Use Areas if the symbols are not meant to correspond to any standardized script, or onto an existing block if the goal is compatibility with an established but rarely supported writing system. Once built, the font loads through `fontspec` exactly as any other custom font from the previous section, and text is entered either directly as the Unicode characters the font maps its glyphs to, or, more conveniently for an author who does not want to memorize a table of private-use code points, through macros that translate a readable, ASCII-based input into the correct underlying Unicode character before typesetting, which is a straightforward application of the macro techniques from Part III to translate a document author's convenient shorthand into the code points the custom font actually expects.

This is a substantial undertaking, appropriate for a project where a distinct visual notation system is genuinely central to the work rather than a nice-to-have, but for a reader who does need it, the path from "I need a symbol no font provides" to "I have typeset text in that symbol" runs entirely through tools already introduced in this book: a font built to contain the glyph, loaded through `fontspec`, and macros to make entering it convenient.

## 10.5 Multilingual Documents

A document mixing multiple scripts, whether a Latin-script main text with occasional Greek or Cyrillic terms, or a genuinely multilingual document with substantial passages in more than one language, relies on the same Unicode-and-OpenType foundation, with a few additional considerations. `polyglossia`, the multilingual counterpart to the older `babel` package built specifically for LuaLaTeX and XeLaTeX, handles per-language hyphenation patterns, language-appropriate typographic conventions such as quotation mark style and date formatting, and switching between fonts automatically when a passage changes language, provided each language's font is specified through `\setotherlanguage` alongside the main `\setmainfont`.

Right-to-left and bidirectional text, needed for languages such as Arabic and Hebrew, is supported directly by both LuaLaTeX and XeLaTeX at the engine level, handling the interleaving of right-to-left and left-to-right runs within a single line correctly, which is a considerably harder typesetting problem than it appears and was one of the genuine motivations for building engines with native Unicode support in the first place, rather than relying on a system built around a Latin-script assumption from the outset. CJK typesetting, with its own line-breaking conventions distinct from space-delimited Latin text, is supported through dedicated packages (`luatexja` for LuaLaTeX, or `xeCJK` for XeLaTeX) that handle these conventions correctly once the appropriate CJK font is loaded.

A document combining several of these considerations at once, a mixed-script page with Latin body text, quoted passages in a right-to-left language, and a technical term rendered in a custom notation font from earlier in this chapter, is demanding to set up correctly, but every piece of that setup traces back to the same underlying model: Unicode code points, OpenType fonts mapped to them through `fontspec`, and an engine, LuaLaTeX or XeLaTeX, built from the ground up to handle Unicode text rather than treating it as an extension bolted onto a Latin-script core.

The next chapter returns to more conventional territory, mathematics beyond the amsmath basics, before Part V turns to managing bibliographies and reference material at the scale a book-length project actually requires.

# Chapter 11

# Mathematics Beyond the Basics

Most LaTeX users learn enough mathematics typesetting to write an equation and stop there: `$...$` for inline material, `\begin{equation}` for a displayed one, a handful of symbols from `amsmath`. A book with a genuine mathematical apparatus, definitions, theorems, proofs, and a consistent notation carried across hundreds of pages, needs more structure than that, not because the underlying commands are more complicated but because the structure has to be designed rather than assembled piecemeal as each new chapter happens to need something.

## 11.1 A Consistent Theorem System

`amsthm` provides the machinery for defining theorem-like environments: `\newtheorem{theorem}{Theorem}` declares an environment that numbers and labels its contents automatically, and the same mechanism extends to definitions, lemmas, propositions, corollaries, remarks, and examples, each as its own environment with its own counter. The choice that matters most, and the one worth making once at the start of a project rather than adjusting chapter by chapter, is how these counters relate to each other and to the chapter structure.

Sharing a single counter across theorems, definitions, and lemmas, so that "Definition 3.4" is immediately followed by "Theorem 3.5" rather than each type of statement keeping its own independent count, makes it possible for a reader to locate any numbered statement by its position in the document without needing to know which type of statement it was. This is achieved by declaring the later environments to share the first one's counter, `\newtheorem{lemma}[theorem]{Lemma}`, rather than giving each its own. For a project as large as a full book, this is close to essential; a reader trying to find "Lemma 12" in a document where lemmas, theorems, and definitions all count independently has no way to search for it without already knowing roughly where it appears.

Resetting this shared counter at each chapter boundary, `\newtheorem{theorem}{Theorem}[chapter]`, so numbering restarts as "Theorem 3.1" rather than continuing an unbroken count from the start of the book, keeps numbers legible and stable under revision: inserting a new theorem partway through Chapter 5 only renumbers the rest of Chapter 5, not every subsequent chapter in the book, which matters considerably for a document under active, ongoing revision where inserting material is common.

## 11.2 Custom Theorem Styles

`amsthm`'s `\theoremstyle` command controls the visual presentation of a theorem-like environment independent of its numbering: `plain` for the traditional italicized body most readers associate with formal mathematics, `definition` for upright text more appropriate to a definition or example, and `remark` for a lighter, less formal presentation suited to asides and notes. A project can and generally should use more than one style, matching the visual weight of a statement's presentation to its actual role: a central theorem set in the more emphatic `plain` style, a routine definition in the calmer `definition` style, so that a reader's eye can distinguish the two categories of content before reading a single word of either.

Beyond the three built-in styles, a custom style defined with `\newtheoremstyle` allows control over spacing, the font used for the statement's body, whether the heading and body run together or the heading stands on its own line, and how the theorem's number is formatted and punctuated. A boxed presentation, wrapping a theorem's content in a visible border, is not part of `amsthm` directly but is straightforwardly layered on top of it using `mdframed` or `tcolorbox` around a theorem environment, giving central results in a book real visual weight on the page, distinct from the plainer treatment appropriate to routine lemmas supporting a larger argument. The right choice between boxed and unboxed presentation is a document-design decision as much as a technical one: a book that boxes every single theorem-like statement loses the ability to use boxing to signal genuine importance, since nothing stands out if everything is emphasized equally.

## 11.3 Custom Operators, Delimiters, and Notation Discipline

Chapter 10 covered the mechanics of defining a new operator or symbol cleanly, avoiding collision with existing definitions. This section is about the design discipline that should govern which symbols get defined in the first place, for a project whose mathematics recurs across many chapters or many books.

A notation system chosen once, early, and used consistently thereafter is worth the up-front effort of deciding it carefully, because changing notation partway through a long project is expensive: every prior use has to be found and updated, and any reader already familiar with the earlier notation has to relearn the replacement. Deciding, before drafting begins in earnest, what symbol represents a recurring concept, what delimiters are used for what kind of grouping, and what conventions govern subscripts and superscripts across the whole project avoids the far more painful alternative of discovering three different ad hoc notations for the same idea scattered across different chapters, each introduced locally by whichever chapter happened to need it first without checking what the rest of the book was already doing.

`\DeclarePairedDelimiter`, from `mathtools`, is worth knowing specifically for this kind of disciplined notation design: it defines a delimiter pair, such as a norm or an absolute value, as a single command that automatically produces correctly sized delimiters (`\left`/`\right`-scaled) without the author needing to write that scaling by hand at every use, and provides a starred variant for the cases where automatic sizing is not wanted. A project defining several custom delimiter-based notations benefits from using this consistently rather than hand-writing `\left\lVert \cdot \right\rVert` at every occurrence, both for the reduced

typing and, more importantly, for the guarantee that every use of that notation is formatted identically throughout the book.

## 11.4 Commutative Diagrams

`tikz-cd`, built on top of TikZ, is the standard tool for typesetting commutative diagrams: objects connected by labeled arrows, arranged in a grid, with support for the full range of arrow styles (injections, surjections, dashed arrows for existence claims, double-headed arrows for natural transformations) that this kind of diagram conventionally uses. For any project whose mathematics involves diagram chasing, functor squares, or any argument more naturally expressed as a diagram than as a sequence of equations, `tikz-cd` is worth learning directly rather than avoided in favor of describing the same content in prose, since a diagram genuinely does convey certain relationships, particularly the commutativity of multiple paths between the same two objects, more clearly and more compactly than an equivalent paragraph of text could.

The syntax is compact once learned: `\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "h"] \\ C \arrow[r, "k"'] & D \end{tikzcd}` produces a two-by-two commutative square with each arrow labeled, and extending this to larger diagrams is a matter of adding rows and columns to the same grid rather than learning new syntax. For a book that uses diagrams of this kind more than occasionally, defining a small set of project-level conventions, consistent arrow styles for consistent relationships, a consistent convention for where labels sit relative to their arrows, pays off the same way notation discipline does elsewhere in this chapter: a reader who has seen one of the book's diagrams can read the rest without relearning its visual vocabulary each time.

With a full mathematical apparatus in place, Part V turns to a different kind of scale problem: managing bibliographies, indexes, and glossaries across a project large enough that keeping any of them consistent by hand is no longer realistic.

# Part V

# Bibliographies and Reference at Scale

# Chapter 12

# Bibliographies at Scale

A short paper's bibliography is rarely a design problem: a dozen entries, added as they come up, checked by eye before submission. A book-length project, or a research program spanning many books and essays over years, is a different problem entirely. This chapter is about choosing the right bibliography tooling for a project's actual distribution needs, and about the discipline required to keep a large reference list consistent as it grows.

## 12.1 Three Ways to Manage a Bibliography

LaTeX offers three broadly different approaches to producing a bibliography, and the right choice depends less on which is more powerful in the abstract and more on what a given project actually needs.

`thebibliography`, the oldest and most basic approach, is a plain LaTeX environment in which each entry is written out by hand, formatted exactly as it should appear, with a `\bibitem` key used for citation. It requires no external tool, no auxiliary compilation step beyond LaTeX itself, and no database file separate from the document's own source. Its cost is that formatting is manual: changing a citation style means reformatting every entry by hand, and there is no automatic mechanism ensuring an entry is only included if it is actually cited, or excluded if it is not.

BibTeX and its more modern successor Biber take the opposite approach: references live in a separate `.bib` database, in a structured field-based format, and a citation style file determines how that data is formatted into an actual bibliography, entirely automatically, including alphabetization, cross-reference resolution between entries, and consistent formatting applied uniformly across every citation regardless of how many hundred entries the database contains. BibLaTeX, paired with Biber rather than the older BibTeX processor, extends this further with substantially more flexible style customization and better handling of non-English names, dates, and a wider range of source types.

The tradeoff is portability. A document using `thebibliography` is fully self-contained: the bibliography is part of the `.tex` source itself, and anyone who receives that one file, or that one small set of files, can compile the document without needing a separate database file, a specific BibTeX or Biber version, or a particular style file to be available. A document depending on BibLaTeX and an external `.bib` file is not self-contained in the same way;

distributing it as a single, easily shared unit means either bundling the database and style files alongside the `.tex` source, or accepting that the document cannot be rebuilt from the `.tex` file alone.

## 12.2 Choosing Per Project

For a project meant to circulate as an individually distributable PDF, an essay or standalone monograph that someone might download and want to rebuild from source without also needing to track down a separate bibliography database, `thebibliography`'s self-containment is a genuine advantage worth the manual formatting cost, particularly when the reference list is modest, a few dozen entries rather than several hundred. This is the reasoning behind using `thebibliography` as the default for standalone essay-scale work: the document's portability matters more than the convenience BibTeX would add for a reference list small enough to format by hand without much burden.

For a project with a genuinely large reference list, several hundred entries or more, reused with overlapping citations across multiple related documents, the calculus shifts. Reformatting hundreds of hand-written entries by hand whenever a citation style needs to change is no longer a minor cost, and the risk of formatting drift, one entry accidentally styled differently from the rest after months of incremental additions, grows with the size of the list. BibLaTeX and Biber's automatic, uniform formatting becomes worth its portability cost at this scale, especially if the project already accepts that it will be distributed as a repository rather than a single file, in which case bundling a `.bib` database alongside the source is not actually a meaningful loss of portability at all.

The right answer, in other words, is not a single project-wide default but a threshold: for small, standalone, portability-first documents, `thebibliography`; for large, actively maintained reference lists shared across a growing body of related work, BibLaTeX and Biber. A single overarching research program can reasonably use both, choosing per document based on that document's own scale and distribution needs rather than forcing every document in the corpus into the same bibliography tooling regardless of fit.

## 12.3 Managing Thousands of References Without Losing Consistency

A `.bib` database that has grown over years, accumulating entries added at different times by different processes, is prone to a specific kind of decay: duplicate entries for the same source under two different keys, inconsistent field formatting (a journal name abbreviated one way in one entry and spelled out in another), and stale entries for sources that were once cited somewhere but no longer are anywhere in the current corpus.

Duplicate detection is worth running periodically rather than only when a duplicate is noticed by accident; tools built for this purpose compare entries by title, author, and year similarity rather than requiring an exact key match, catching the case where the same paper was added twice under two different citation keys because whoever added the second entry did not realize the first already existed. Consistent field formatting is best enforced at the point of entry rather than corrected after the fact: a standard template for how a journal

article, a book, and a conference paper are each entered, followed consistently, prevents the formatting drift that is otherwise expensive to clean up retroactively across a database with hundreds of entries already in it. Stale-entry cleanup, removing entries no longer cited by anything in the current corpus, is lower priority than the other two, since an unused entry causes no visible problem, but a periodic pass checking which keys in the database are actually referenced by any document in the corpus keeps the database itself from growing indefinitely with dead weight.

## 12.4 Cross-Document Citation Strategy for a Multi-Book Corpus

A research program spanning many related books and essays, citing many of the same sources repeatedly across different documents, benefits from treating its bibliography the same way Chapter 10 recommended treating a shared notation system: as a maintained, shared resource rather than something reinvented per document. A single master `.bib` database, included by any document in the corpus that uses BibLaTeX, ensures the same source is cited under the same key and formatted the same way everywhere it appears, rather than drifting into slightly different entries for the same work across different books.

For documents using `thebibliography` instead, the equivalent discipline is a maintained reference block of commonly cited sources, copied into a new document's bibliography as a starting point rather than retyped from memory, with the same formatting preserved across documents even though there is no shared database file enforcing that consistency automatically. This is more manual than the BibLaTeX approach, but it is exactly the tradeoff already accepted in choosing `thebibliography` for portability, and the manual discipline of copying from a maintained reference block rather than retyping is what keeps that tradeoff from costing more than it needs to.

Self-citation is worth a specific note in a corpus this large: a growing body of interlinked essays and books naturally wants to reference the author's own earlier work, but a bibliography padded primarily with self-citation reads as inward-looking regardless of how relevant the citations actually are, and is worth avoiding as a matter of editorial discipline independent of any technical bibliography-management question. Where an earlier work in the corpus is genuinely the right source for a claim, referencing it in prose, by title, is usually more appropriate for this kind of project than a formal bibliography entry, which is why the recommended convention throughout this book's own practice is to reference the corpus in this way rather than build a bibliography that cites itself.

The next chapter turns to indexes, glossaries, and nomenclature, the remaining pieces of a book's back-matter apparatus, before Part VI moves on to graphics and computational automation.

# Chapter 13

# Indexes, Glossaries, and Nomenclature

A book's index and glossary are the parts of its back matter readers consult rather than read straight through, and precisely because of that, they are easy to treat as an afterthought bolted on once the main text is finished. For a project with its own recurring, specific vocabulary, built up across many chapters or many books, treating index and glossary construction as an afterthought produces exactly the kind of inconsistency that makes them least useful: the same concept indexed under two different terms, or a glossary that defines a term without covering every sense the book actually used it in.

## 13.1 Building an Index with `makeindex`

An index entry in LaTeX is marked at the point in the text it applies to, using `\index{term}`, and the actual sorted, page-numbered index is generated by a separate tool, `makeindex`, run on the raw index data LaTeX writes out during compilation, in the same multi-pass pattern Chapter 5 described for bibliographies: LaTeX writes raw entries, `makeindex` processes and sorts them, and a subsequent LaTeX pass reads the processed result back in and typesets it.

`\index` entries support subentries (`\index{notation!custom operators}`, producing a nested index entry under "notation"), cross-references (`\index{notation|see{symbols}}`), and explicit sort keys for terms that should be alphabetized differently from how they are typeset (`\index{TeX@\TeX}`, ensuring a command like `\TeX` sorts under T rather than by its literal, unsortable macro name). For a technical book with its own recurring vocabulary, subentries are worth using deliberately rather than only when a term happens to accumulate enough page references to need them: grouping related terms under a shared parent entry from the start makes the index itself into a small map of the book's conceptual structure, not merely an alphabetized list of page numbers.

The discipline that actually determines whether an index is useful is consistency at the point of marking, not anything `makeindex` itself can enforce. A concept indexed as "admissibility" in one chapter and "admissible states" in another produces two separate index entries for what a reader experiences as one concept, and no tool catches this automatically;

it has to be caught by maintaining a short list of the book's key indexed terms, exactly as Chapter 10 recommended maintaining a shared notation file, and checking new `\index` entries against that list rather than inventing a phrasing fresh at each point in the text.

## 13.2 Glossaries and Nomenclature with the `glossaries` Package

The `glossaries` package handles a related but distinct need: defining a term once, with a full definition, and then referencing it throughout the text in a way that can produce both an inline reference and a collected glossary at the back of the book. A term is declared once, typically in the shared preamble file described in Chapter 3, with `\newglossaryentry{admissibility}{name={ad` `description={...}}`, and referenced in the body with `\gls{admissibility}`, which typesets the term itself while also recording that this location used it, feeding into the automatically generated glossary the same way indexed terms feed into the index.

`glossaries` additionally supports first-use behavior distinct from subsequent uses, useful for acronyms in particular: `\gls` on an acronym's first use in the document can expand to the full term followed by the abbreviation in parentheses, while every subsequent use renders as the abbreviation alone, without the author needing to track by hand which use is the first. For a book defining a genuinely technical vocabulary rather than only abbreviations, the same first-use mechanism can be used more broadly, so a term's first appearance in each chapter, or in the book as a whole depending on configuration, gets slightly more context than later uses that can now rely on the reader already having encountered the definition.

A nomenclature list, distinct from a glossary of terms, is more appropriate for a book with heavy recurring mathematical notation: rather than defining what a word means, it lists what a symbol means, typically organized by the order symbols were introduced or grouped by category (Latin letters, Greek letters, custom operators). The `nomencl` package, or `glossaries`'s own support for symbol-type entries, both handle this, and for a project already maintaining the shared notation file recommended in Chapter 11, generating a nomenclature list from that same file, rather than maintaining the list of symbols separately, keeps the nomenclature from drifting out of sync with the notation actually used in the text.

## 13.3 When a Glossary Earns Its Keep

Not every book needs a glossary, and adding one reflexively to any technical project produces the same kind of clutter an unnecessary index would: a glossary of five terms a reader could infer from context on first encounter adds a page of back matter without adding proportional value, while a genuinely large, recurring technical vocabulary, the kind a reader is likely to need to look back up multiple times across a long book, is exactly the case a glossary is built for.

A reasonable test, worth applying deliberately rather than defaulting either way, is whether a term is likely to recur across chapters written far enough apart, in drafting time, that a reader (or the author, months later) would plausibly want to check its definition again without rereading the chapter that first introduced it. Terms that clear this bar belong in

a glossary; terms defined and used entirely within a single chapter, never referenced again, are usually better served by a definition environment at the point of use, as covered in Chapter 11, than by a glossary entry that would only ever be consulted from the one place it is used.

## 13.4 Nomenclature Across a Multi-Book Corpus

For a research program spanning many books that share notation, as Chapter 10 and Chapter 11 both recommended treating symbol definitions as a maintained cross-corpus resource, the nomenclature list built for any one book benefits from being generated from that same shared resource rather than compiled independently per book. A symbol's meaning that stays consistent across the corpus should produce an identical nomenclature entry wherever it appears, and generating that entry automatically from the single shared notation file, rather than transcribing it by hand into each new book's own nomenclature list, is both less work and the more reliable way to keep a large body of related books actually consistent with each other over years of continued writing.

Part V closes here, with the reference apparatus, bibliography, index, glossary, and nomenclature, all treated as maintained resources rather than one-off additions. Part VI turns to graphics and automation: bringing figures, generated plots, and computation itself directly into the publishing pipeline this book has been building toward since Chapter 3.

# Part VI

# Graphics and Automation

# Chapter 14

# Graphics as a Pipeline, Not an Attachment

The default way most people put a figure into a LaTeX document is `\includegraphics`, pointing at an image file produced somewhere else, by some other program, and pasted in much the way a figure would be pasted into a word processor. This works, and for a photograph or a hand-drawn illustration it is the only sensible approach, since there is no meaningful sense in which such an image could be generated from the document's own source. For a plot, a diagram, or any figure that represents data or a formal structure the document itself already describes, treating the figure as an external attachment throws away one of LaTeX's genuine advantages: the ability to define the figure in the same source language as the surrounding text, so that regenerating the document also regenerates the figure, correctly, from whatever the current state of the underlying data or argument actually is.

## 14.1 TikZ for Figures Defined in the Document

TikZ is a graphics language embedded in LaTeX itself, letting a figure be specified as LaTeX source rather than as a separate binary image file. A diagram drawn in TikZ lives in the same file as the text that discusses it, uses the same fonts as the surrounding document automatically, since it is typeset by the same engine rather than imported as a pre-rendered image, and can reference the document's own macros and notation directly, so a diagram's labels stay consistent with the notation used in the prose around it without any separate effort to keep the two synchronized.

This matters most for diagrams that need to change as an argument develops. A hand-drawn or externally-produced diagram that needs updating requires opening a separate tool, making the edit, re-exporting, and re-importing; a TikZ diagram is edited exactly like any other part of the document's source, and its update is included in the same commit, the same diff, and the same build as everything else, which keeps a diagram from silently going stale relative to a revised argument the way an externally maintained image file easily can. For any figure that is fundamentally schematic, an architecture diagram, a state machine, a commutative diagram as covered in Chapter 11, a flowchart describing a process the

surrounding text also describes, TikZ is the tool that keeps the figure and the argument it illustrates from drifting apart over the course of a long revision history.

## 14.2 PGFPlots and Data-Driven Figures

PGFPlots, built on top of TikZ, extends this same in-document approach to actual plots: line graphs, scatter plots, bar charts, and surface plots, specified either with inline coordinate data or, more usefully for a figure that represents real data rather than a hand-constructed illustration, by reading directly from an external data file at compile time. `\addplot table {data.csv};` pulls plotting data straight from a CSV file, meaning the figure updates automatically the next time the document is built, if the underlying CSV file has changed, with no manual step of re-exporting a plot image and re-importing it.

This is the concrete mechanism behind the principle this chapter opens with: a figure built this way is not attached to the document, it is derived from the same data the document's own build process already has access to. A plot showing experimental results stays current with the latest data automatically; a plot showing results from an earlier stage of an analysis that has since been revised will visibly and correctly update the next time the document is rebuilt, rather than silently continuing to show stale results because nobody remembered to regenerate and re-import a separate image file.

## 14.3 CSV-to-Table Workflows

The same principle applies to tables as to plots, and is arguably even more valuable there, since a hand-transcribed table is one of the most common sources of a document silently diverging from its actual underlying data. `pgfplotstable`, part of the PGFPlots ecosystem, reads a CSV file and typesets it as a properly formatted LaTeX table directly, with support for column-specific number formatting, computed columns derived from the source data, and filtering or sorting applied at typesetting time rather than requiring the CSV itself to already be in exactly the right order and format.

For a book presenting any substantial amount of tabular data, results tables, comparison tables, parameter listings, generating these directly from the CSV or other structured data file that is the actual source of truth, rather than transcribing values into a hand-written `tabular` environment, removes an entire category of error: a hand-transcribed table can silently diverge from its source the moment the source is updated and the table is not, while a table generated at compile time from the same file cannot diverge from that file by construction, since it is reading it fresh on every build.

## 14.4 Computation Outside LaTeX, Included Inside It

Not every figure's data originates in a format PGFPlots or `pgfplotstable` can read directly, and not every plot is simple enough to be worth expressing in PGFPlots's own plotting language rather than a dedicated plotting library in Python, R, or Julia. For these cases, the pipeline principle still applies, just with an external step made explicit rather than avoided: a script, checked into the project alongside the document's own source as described in

Chapter 4, reads the underlying data, produces a plot as a PDF or SVG, and the document includes that output with `\includegraphics` exactly as it would any other image, with the crucial difference that the generating script is part of the project, versioned alongside everything else, and rerunning it is a documented, repeatable step rather than a one-off action performed once and then forgotten.

Wiring this generation step into the build process itself, through a Makefile target as described in Chapter 5 that runs the plotting script before invoking `latexmk`, or by making the figure a dependency the Makefile knows how to regenerate whenever its source data changes, closes the remaining gap between this hybrid workflow and the fully in-document approach of TikZ and PGFPlots: even when the actual plotting happens outside LaTeX, "rebuild the document" can still mean "regenerate every figure from current data and then typeset," rather than "typeset whatever figure files happen to already be sitting in the figures directory," which is precisely the guarantee that makes a figure trustworthy months or years after it was first produced.

The next chapter extends this same pipeline further, bringing computation more directly inside the document itself: embedded code listings, verified output, and simulation results woven into the text as a single build rather than a document written up separately from the computation it describes.

# Chapter 15

# Computation Inside the Document

The previous chapter treated figures as derived data rather than attachments. This chapter extends the same principle to computation more broadly: code listings that are verified rather than merely typed, simulation output that regenerates automatically, and a build process where every commit produces a document that is guaranteed to reflect the current state of the underlying computation, not a snapshot someone remembered to update the last time they happened to rerun it by hand.

## 15.1 Verified Code Listings with `minted`

Most LaTeX documents that include source code do so with `verbatim` or `listings`, both of which typeset whatever text is given to them without checking that it is valid, runnable code in the language it claims to be. `minted`, built on the external Pygments syntax highlighter, provides considerably better highlighting across a wide range of languages, but its more important property for a technical book is what it makes possible in combination with a build pipeline: because `minted` can also execute code through Pygments' broader ecosystem tooling, or be paired with a build step that runs the listed code separately and compares its output against what the document claims that output to be, a code listing can be checked rather than merely displayed.

This distinction matters most for a book, like this one, whose worked examples are meant to actually work. A hand-typed code listing that once ran correctly, at the time it was written, can silently drift out of sync with the library or language version it was written against, and nothing about a plain `verbatim` block catches this drift; the listing continues to render exactly as typed regardless of whether it would still run. A listing pulled from an actual source file checked into the project, included with `\inputminted{python}{scripts/example.py}` rather than retyped inline, cannot drift from what that file contains, and if the project's build process also runs that file as part of its test suite, a change that breaks the example is caught at build time rather than silently shipped in the next PDF.

## 15.2 Calling Out to External Languages at Build Time

For simulation results, numerical examples, or any computed value a book wants to report precisely, retyping a number by hand from wherever it was computed is the same category of error as hand-transcribing a table from a CSV file, covered in the previous chapter: the number in the document can silently diverge from what the underlying computation actually produces the moment either one is updated without the other. The more robust pattern, extending the Makefile-driven approach from Chapter 14, is a build step that runs the relevant Python, Rust, MATLAB, R, or Julia code, writes its output, whether a number, a table, or a figure, to a file the document then reads, and rebuilding the document means rerunning that computation first.

For a single computed value referenced inline in prose, this can be as simple as a script that writes a small `.tex` file defining a macro (`\newcommand{\resultvalue}{4.7}`), included into the main document, so that `\resultvalue` in the prose always reflects the most recently computed result rather than a number someone typed once and never revisited. For more substantial output, a full table or figure, the same CSV-based pipeline from the previous chapter applies directly, with the computation itself, not merely its formatting, as the first step of the pipeline rather than something that happened once, off to the side, before the document's own build process began.

## 15.3 The Document as the Final Stage of a Computation

Adopting this pattern consistently changes what a "build" of the document actually means. Rather than a document being written up after a computation is finished, with numbers and figures copied over by hand at whatever point the writing catches up to the results, the document's own build process becomes the last stage of the computation itself: run the analysis, generate the figures and computed values, then typeset the document that presents them, all as one sequence triggered by a single command.

This has a consequence worth stating directly, because it changes how a document should be trusted. A document built this way cannot present a stale result without the build itself failing to actually rerun the computation, which is a detectable, fixable failure of the build process rather than a silent, undetectable failure of an author's memory to update a hand-copied number. A reviewer, or a reader building the project from source months after publication, gets the same guarantee: the numbers in front of them are what the checked-in code actually produces, right now, on this machine, not what it produced once, on the author's machine, at some earlier point that may no longer match the current state of the code.

## 15.4 Continuous Integration: A Fresh PDF on Every Commit

The natural extension of this discipline is running the entire pipeline, computation and typesetting together, automatically on every commit, rather than only when an author remembers to run it locally before sharing a draft. A continuous integration workflow,

using GitHub Actions or a comparable service, checked into the repository alongside the document's own source, can be configured to install the project's dependencies (the TeX distribution, the plotting and simulation libraries, any language runtimes the build scripts require), run the full build, computation and all, and publish the resulting PDF as a build artifact attached to that commit.

This closes the loop the earlier sections in this chapter opened. A broken example, a computation that no longer runs against a current library version, or a figure-generating script that has drifted out of sync with the data it expects, is caught automatically, on every commit, rather than discovered by a reader months later or by the author themselves the next time they happen to rebuild the project from scratch. For a book-length project under active development for years, where memory of exactly how each figure was originally generated fades long before the project is finished, this is not a convenience; it is what keeps a large, computationally grounded document trustworthy over a timescale longer than any single person's memory of their own build process can be relied on to cover.

Part VI closes here, with graphics and computation both treated as part of the document's own build rather than as external inputs prepared separately and pasted in. Part VII turns to document engineering proper: custom document classes, professional title pages and front matter, publishing pipelines, and debugging a large project when, despite all of this discipline, something still goes wrong.

# Part VII

# Document Engineering

# Chapter 16

# Custom Document Classes

Every LaTeX document begins with \documentclass, and for most users the choice is limited to whichever standard class, article, report, book, happens to be closest to what they need, patched afterward with enough preamble packages and redefinitions to approximate the actual desired result. For a single document, this is a reasonable way to work. For a research program producing many related books over years, each one wanting the same title-page design, the same front-matter structure, the same theorem styles and notation conventions, patching a standard class identically in every new document's preamble is both repetitive and fragile: any improvement to the shared conventions has to be copied by hand into every existing document's preamble to take effect there too, and inevitably some copy falls out of sync with the rest.

A custom document class solves this the way a shared library solves the equivalent problem in software: the conventions live in one place, a .cls file, and every document that uses them does so by loading that one file, so an improvement made once is available everywhere the class is used, and every document built with a given version of the class is guaranteed to share its conventions exactly, rather than approximately.

## 16.1 Anatomy of a .cls File

A document class file is not fundamentally different from any other body of LaTeX macros; what distinguishes it is that it is loaded through \documentclass rather than \usepackage, and by convention it is responsible for loading a base class underneath it, typically book or report for a project of the scale this book has been discussing, via \LoadClass. A minimal custom class might do nothing more than load book, set the page geometry, load fontspec and set the project's standard typeface, and load the packages (amsthm, mathtools, hyperref) every document in the project needs regardless of content, replacing what would otherwise be a preamble block copied identically into every book's top-level file with a single \documentclass{myresearchclass} line.

Beyond this minimal case, a class file can define new commands specific to the project's document type, redefine how existing structural elements like chapter headings or the table of contents are typeset, and declare class-specific options, \documentclass[draft]{myresearchclass} versus \documentclass[final]{myresearchclass}, that change the class's behavior at

load time, using \DeclareOption and \ProcessOptions to parse whatever options a specific document invocation passes in. This options mechanism is what lets a single class file serve every book in a series while still allowing per-document variation, a draft watermark, a different page size for a print edition, without those variations requiring a different class file for every case that needs them.

## 16.2 Enforcing Front Matter at the Class Level

The most direct payoff of a custom class, for a project maintaining consistent title pages and front matter across many books as discussed later in this part, is the ability to require and structure metadata at the class level rather than leaving it to each document's own ad hoc preamble. A class can define its own commands, \bookauthor, \bookaffiliation, \bookdate, distinct from the generic \author and \date the base class already provides, giving the class author full control over exactly what fields a title page requires and exactly how they are formatted, rather than working within whatever \maketitle the base class happens to already implement.

Where a base class's \maketitle is difficult to adapt without extensive redefinition, because it was not designed with a particular title-page layout in mind, a custom class can simply define its own title-page command entirely, \makebooktitle, built from scratch to lay out exactly the fields the project's title pages should contain, in exactly the arrangement the project has settled on, with any field the class considers mandatory checked for presence, and an author-facing warning or error raised at compile time if a required field, an affiliation, a required copyright notice, is missing rather than that omission only being caught by a human proofreading the rendered title page after the fact.

This is worth treating as a genuine specification, not merely a convenience: a class-level requirement that every document declares its title, author, and date before \makebooktitle can run is a guarantee, enforced by the compiler itself, that no document built with this class can accidentally ship without that information, which is a stronger guarantee than any style guide or personal checklist can provide on its own.

## 16.3 One Class, Many Books

For a project maintaining several related book-length works, a single shared class file, versioned in its own repository or as a shared submodule included by each book's own repository, is the mechanism that keeps title-page design, notation conventions, and theorem styles genuinely identical across the whole series rather than merely similar. A change to the class, adjusting the title-page layout, adding a new theorem style, updating the standard bibliography formatting, propagates to every book the next time that book is rebuilt against the updated class, without requiring the change to be manually reapplied to each book's own preamble.

This does introduce a dependency-management question a single-document project does not have: which version of the shared class does each book in the series actually depend on, and what happens when the class changes in a way that would alter how an already-published book renders. The same discipline Chapter 4 recommended for reproducibility

generally applies here directly: pinning each book's build to a specific, recorded version of the shared class, whether through a Git submodule at a fixed commit or an explicit version note in the book's own build documentation, so that rebuilding an earlier book from source still produces the same output it originally did, even after the shared class has continued to evolve for newer books in the series.

The next chapter turns to what a custom class like this is often built specifically to support: professional title pages, front and back matter conventions, and the print-ready output requirements a book intended for real publication, rather than only PDF distribution, actually needs.

# Chapter 17

# Title Pages, Front Matter, and Print-Ready Output

A title page is the first thing any reader sees, and it is also, for exactly that reason, one of the last things an author actually finalizes. This chapter covers the design and technical mechanics of professional front matter and print-ready output, but it opens with a tension worth naming directly, because it shapes how the rest of the chapter's advice should be applied rather than being a side issue to mention and move past.

## 17.1 A Genuine Contradiction: Early Metadata, Late Titles

LaTeX's structure asks an author to declare a document's title, author, and date near the top of the source, in the preamble, before a single word of the actual content has been written, using `\title`, `\author`, and `\date`, or the class-level equivalents from the previous chapter. This placement is not arbitrary; the preamble is also where packages are loaded and where the document's typographic identity, fonts, theorem styles, notation, is fixed, and a class expecting required front-matter fields, as the previous chapter recommended, naturally wants those fields declared early, in the same place as everything else the document depends on structurally.

The difficulty is that a book's actual title is often one of the last things to stabilize, not the first, particularly for a long-form work whose central argument develops and sometimes shifts meaningfully over the course of drafting. A working title chosen at the outset frequently turns out, by the time a manuscript is finished, to describe an earlier and narrower version of the book than what was actually written, and retitling at that point is not merely a matter of changing the string inside `\title{}`. A title change of any real consequence tends to reflect, or force, a change in emphasis that ought to be felt throughout the book: which chapter gets foregrounded in the preface, which thread gets called the central argument in the introduction and which gets recast as supporting material, occasionally even which appendix material deserves promotion into the main text now that the book's own sense of what it is about has shifted.

There is no purely technical resolution to this, because the difficulty is not really about where `\title{}` sits in the file. It is a genuine mismatch between LaTeX's front-loaded

declaration of a document's identity and the fact that a book's identity, in the sense that actually matters, is frequently not settled until close to the end of writing it. The practical response is to treat the title declared early as provisional by default for any project expected to take more than a few weeks, rather than as a commitment made in the preamble and then forgotten about; and, more importantly, to treat a late title change as a prompt to reread the introduction and preface specifically for passages that were written to support the old title's framing rather than the new one, since those are exactly the passages most likely to have drifted out of alignment with what the book, retitled, is now claiming to be about. A title change that touches only the preamble and nothing else in the manuscript is a signal worth treating with some suspicion: it may mean the new title is genuinely a surface relabeling with no deeper implications, but it may also mean the rest of the book has not yet caught up to what the new title is promising a reader.

## 17.2 Designing a Title Page

With that caution in mind, the mechanics of a title page itself are straightforward once a custom `\makebooktitle`, as described in the previous chapter, is in place. A title page that reads as professionally published rather than templated typically uses restrained typography, the same type family as the body text rather than a decorative display face reserved only for this one page, generous whitespace rather than a page crowded with every possible piece of metadata, and a clear typographic hierarchy: the title set larger and with more weight than the subtitle, the subtitle larger than the author line, the author line larger than the affiliation. Centering everything by default, without considering whether a left-aligned or asymmetric layout might better suit the specific book, is one of the most common tells of an untailored title page; the choice should be made deliberately, matching the visual conventions established by the rest of the book's design, rather than defaulted to.

## 17.3 Front Matter and Back Matter Conventions

Front matter, the material preceding the main text proper, conventionally includes a half-title page, the full title page, a copyright and edition page, a dedication where relevant, a table of contents, and, where present, lists of figures and tables generated automatically from the document's own float captions rather than maintained by hand. A preface, distinct from an introduction, addresses the reader about the book itself, its motivation and scope, while an introduction begins the book's actual argument; conflating the two, or omitting the preface entirely for a book that would benefit from directly addressing why it exists before diving into its subject matter, is a common structural gap in self-published technical work that a deliberate front-matter plan avoids.

Back matter, following the main text, conventionally includes appendices, a bibliography, an index, a glossary or nomenclature list where the project maintains one as described in Chapter 13, and, for a book meant to read as a complete, finished object, a colophon: a short closing note on how the book was produced, the typeface used, the engine it was built with. Placed correctly, a colophon belongs entirely in the back matter, never on the title page itself; a stray production note or build instruction accidentally left on a title page is

not merely an aesthetic slip; automated tools that ingest and summarize documents have been observed treating such a note as though it were meaningful content of the book rather than production metadata, which is reason enough to keep this material strictly confined to its proper place in the back matter.

## 17.4 Cover Design and Print-Ready PDFs

A PDF intended only for screen reading has different requirements than one intended for print, and conflating the two produces a file that looks fine on a monitor and fails, sometimes expensively, at a print shop. Print-ready output requires bleed, extra content extending past the trim line on any page with full-bleed color or imagery, so that minor variation in the physical cutting process does not leave a visible white sliver at the edge; a defined trim size matching the exact physical dimensions the printer expects, distinct from the page size a screen-reading PDF would use; and a color profile appropriate to the print process, typically CMYK for offset printing rather than the RGB most digital tools default to, since colors that look correct in RGB on a screen can shift unpredictably when converted to CMYK ink without deliberate profile management.

Font embedding is the other requirement that print workflows enforce far more strictly than screen-reading ones: every font used anywhere in the document, including any custom or licensed font from Chapter 10, has to be fully embedded in the final PDF, not merely referenced by name, since a print shop's rendering pipeline has no access to fonts installed only on the author's own machine. LuaLaTeX and XeLaTeX both embed fonts automatically as part of ordinary PDF generation, but confirming this with a PDF inspection tool before sending a file to print, rather than assuming it happened correctly, catches the rare case where a font's own licensing metadata restricts embedding in a way that silently produces a PDF referencing a font that will not actually be present at the print shop's end.

Cover design itself is typically produced separately from the interior typesetting, in a dedicated design tool rather than LaTeX, since a wraparound cover with spine width dependent on final page count is a fundamentally different kind of layout problem than interior typesetting; LaTeX can compute an accurate final page count for exactly this purpose, `\pageref{LastPage}` from the `lastpage` package, which is often the one piece of information a cover designer actually needs from the interior's own build process before finalizing spine width.

With title pages, front matter, and print-ready output covered, the next chapter turns to publishing pipelines proper: producing more than one output format, a book, a standalone paper excerpt, a website, from a single maintained source tree.

# Chapter 18

# Publishing Pipelines

The traditional view of publishing treats a document as the end product of writing: text is composed, figures are copied into place, citations are adjusted, and eventually a PDF is produced. This works for a short report and scales poorly beyond it, because every output under this model is an independent artifact maintained by hand, and consistency across outputs depends entirely on an author remembering to update each one whenever any of them changes. This chapter treats publishing the way the rest of Part VI treated figures and computation: as a pipeline with the source tree, not any single rendered file, as the actual primary artifact.

## 18.1 The Source Tree as the Primary Artifact

Once a project is organized the way Chapter 3 recommends, chapters, figures, bibliography, and notation each as separate, maintained files, the PDF stops being the thing that is authored and becomes one rendering of something else: the repository as a whole. Correcting a theorem, updating a figure's underlying data as in Chapter 14, or fixing a bibliography entry as in Chapter 12 changes the source once, and every output generated from that source reflects the correction automatically the next time it is built, rather than requiring the same fix to be reapplied by hand to a book PDF, a website, and a slide deck independently. This is the same discipline Chapter 15 applied to computation, extended to the publishing process itself: the workflow resembles software engineering considerably more than it resembles desktop publishing.

## 18.2 Markdown as an Interchange Language

A pipeline built this way often begins further upstream than LaTeX itself. Markdown's minimal syntax, emphasizing semantic structure (this is a heading, this is emphasis, this is a list) over presentation detail, makes it convenient for early drafting, for collaborators who do not need to know LaTeX to contribute text, and for version control, since a Markdown diff shows a content change far more legibly than a diff against heavily formatted LaTeX source would. Markdown is not a replacement for LaTeX in this pipeline; it is a convenient,

low-overhead representation for the stage of writing where content is still taking shape and precise typesetting is not yet the point.

## 18.3 Pandoc as the Transformation Engine

Pandoc is what connects these representations. A Markdown manuscript can be translated into LaTeX for typeset output, into HTML for a project website, into EPUB for electronic readers, or into DOCX where a collaborator's workflow requires conventional office software, all generated from the same semantic source rather than maintained as separate, independently edited files. This matters for the same reason a shared bibliography database matters in Chapter 12: a single authoritative source, transformed on demand into whatever format a given audience needs, cannot drift out of sync with itself the way several independently maintained copies inevitably do.

LaTeX occupies the final stage of a pipeline built this way, not because it competes with Markdown but because the two serve different needs: Markdown is suited to authoring and portability, while LaTeX, once a manuscript's content has stabilized enough to warrant it, is suited to the precision, automation, and professional typesetting the rest of this book has been concerned with, cross-references, indexes and glossaries as in Chapter 13, mathematical notation as in Chapter 11, bibliography formatting as in Chapter 12. A superuser chooses the representation appropriate to each stage of a document's life rather than insisting the entire process happen in a single language throughout.

## 18.4 One Repository, Many Outputs

Organized this way, a single repository can produce several genuinely different outputs from the same underlying source: a full monograph typeset through LuaLaTeX per this book's conventions, individual chapters exported as standalone papers, HTML documentation published to a project website, lecture notes assembled from a selected subset of chapters, and presentation slides sharing the same figures and notation as the book they are drawn from. None of these require duplicate authoring, since all of them originate from the same source tree; producing a new output format is a matter of adding a new transformation target to the pipeline, not a matter of rewriting content that already exists elsewhere in a different form.

## 18.5 Content and Presentation as Reusable Components

This architecture rewards treating a document's actual content, definitions, bibliographic references, code listings, diagrams, and datasets, as components maintained once and referenced everywhere they are needed, rather than as text copied between documents by hand. A correction made to a definition in the shared notation file from Chapter 10, or to a dataset feeding a figure as in Chapter 14, benefits every downstream publication built from that source automatically. Publishing organized this way acquires the properties Chapters 3 through 5 already established for the underlying document architecture: modularity, re-

producibility, and, with the CI approach from Chapter 15 extended to publishing itself, automated verification that the whole pipeline still runs correctly after every change.

## 18.6 Automating the Pipeline

The natural conclusion of this architecture is automation at the level of the whole publishing process, not merely the LaTeX build covered in Chapter 5. A continuous integration system, triggered on every commit, can rebuild every output the repository is configured to produce, the monograph, the website, any standalone excerpts, verifying that the full pipeline still runs successfully and publishing updated artifacts without manual intervention. Publishing, treated this way, stops being a separate activity performed after writing is finished and becomes an automated consequence of maintaining a consistent, correctly structured source tree, exactly as Chapter 15 described for computation feeding into a single document, now extended to every format that document is published in.

The guiding principle running through this chapter, and through the book's treatment of graphics, computation, and publishing together across Part VI and this chapter, is that a superuser does not write documents independently of one another. They design a system in which a book, a paper excerpt, a website, and a set of lecture notes are different views of the same underlying, carefully maintained source, so that once that source is treated as the single authoritative representation, every output built from it is reproducible, maintainable, and able to grow to meet a new need, a new format, a new audience, without the accumulated maintenance burden that independently authored outputs would otherwise impose.

The next chapter turns from the pipeline's architecture to what happens when, despite all of this structure, a build still fails: reading log files, tracing macro expansion, and debugging a large project like a superuser rather than by trial and error.

# Chapter 19

# Debugging Like a Superuser

The defining characteristic of a superuser is not that they never encounter errors, but that they know how to interpret them. Most LaTeX users respond to a failed compilation by changing random lines until the error disappears, or worse, by copying fragments from internet forums without understanding why they work. This treats debugging as guesswork. A superuser instead regards the compiler as an instrument reporting the current state of the document, and treats the task not as eliminating an error by any means available, but as understanding why the document entered an inadmissible state and what minimal intervention restores it.

## 19.1 Reading the Log File

The first habit worth cultivating is reading the `.log` file directly rather than relying on an editor's abbreviated error summary. Every compilation produces a detailed account of package loading, macro expansion, font selection, and page breaking, and the message an editor surfaces is often only the final, downstream consequence of a problem that originated much earlier. A missing brace, for instance, frequently does not produce an error at the point it is missing; TeX continues parsing, absorbing subsequent text into whatever group it believes is still open, until it can no longer maintain a consistent interpretation of the input, at which point an error surfaces hundreds of lines later, in a location that has nothing to do with the actual mistake. The `.log` file's complete execution history, read from the point just before the reported error backward, is what actually identifies the original cause; the editor's summary, taken alone, frequently points a debugging effort in the wrong direction entirely.

## 19.2 Expansion Errors versus Runtime Errors

Understanding TeX's expansion model, covered in Part I and Part III of this book, matters directly here, because many apparent runtime failures are in fact expansion failures with a runtime-looking symptom. A document does not fail because some command executed incorrectly in the way a runtime error would in a conventional language; it fails because the wrong sequence of tokens was produced during expansion, before anything resembling

execution began, and the resulting malformed token sequence only produces a visible error once something further downstream tries and fails to make sense of it.

`\tracingmacros` and `\tracingcommands`, set to a nonzero value and written to the log with the aid of `\tracingonline` during a targeted debugging session, reveal this hidden expansion process directly: every macro expansion is recorded as it happens, showing exactly what a macro's arguments were and what it expanded to, which makes it possible to observe precisely where an unexpected token sequence first appeared rather than inferring it indirectly from a downstream symptom. `\show` and `\showthe`, used more surgically at a single point in a document, report a macro's current definition or a register's current value without needing to enable full tracing, and are usually the faster tool for a specific, localized question ("what does this macro currently expand to") rather than a broad investigation. These tools are verbose, and reading their output takes practice, but they are the direct evidence Part I's model predicts should exist, and using them is the difference between confirming a hypothesis about where a problem originates and merely guessing at it.

## 19.3 Profiling Compilation Performance

Performance problems deserve the same systematic treatment as correctness problems. A large monograph, with thousands of pages, hundreds of figures, and an extensive bibliography, frequently compiles slowly enough that the temptation is to accept the slowness as inevitable rather than to investigate it. It is rarely uniform: external graphics, TikZ figures recomputed from scratch on every build, syntax-highlighted listings processed through `minted`'s external Pygments call, glossary and index generation, and bibliography processing each contribute a measurable, separately identifiable cost, and `texloganalyser` or a comparable log-timing tool, alongside simple wall-clock timing of individual build stages through the Makefile targets described in Chapter 5, can identify which of these actually dominates a given project's build time. A project whose TikZ figures are the primary cost benefits from caching compiled figures between builds and only regenerating the ones that changed; a project whose bibliography processing dominates benefits from investigating whether Biber's own caching is being invalidated unnecessarily. Guessing at which stage is slow and optimizing it without measurement frequently targets the wrong stage entirely, spending effort on a part of the build that was never the actual bottleneck.

## 19.4 A Philosophy of Repair: The Nearest Admissible State

The overwhelming majority of compilation failures arise from a small, local inconsistency: an unmatched delimiter, a malformed macro argument, an incorrectly nested environment, a missing package. The document as a whole remains largely valid; only some small part of it has drifted into a state the compiler cannot process. The objective, correctly understood, is to restore the nearest state the compiler can accept, not to reconstruct the project from memory or rewrite substantially more than what actually broke.

This can be stated a little more precisely without turning it into a heavier formal exercise than it needs to be. Treat the space of possible document states as everything the source files could contain, and treat a state as admissible if it compiles to a valid, intended output; a

failed build is a document that has moved, through some edit, from an admissible state into an inadmissible one. Debugging, under this framing, is the search for a repair, a small edit, ideally the smallest one that actually resolves the failure, that returns the document to an admissible state again. This is worth stating explicitly because it reframes what "success" in debugging looks like: the goal is not any edit that makes the error message disappear, since a sufficiently large or careless edit can often suppress an error while introducing a different, sometimes less visible, problem elsewhere; the goal is the minimal repair, the smallest change consistent with removing the actual inconsistency, because that is the change most likely to restore the document to what it was actually meant to say, rather than to some other admissible state that happens to compile but no longer matches the author's intent.

## 19.5 Version Control as a Search Procedure

This framing turns directly into a practical technique once a project follows Chapter 4's version control discipline: if the document compiled successfully at some earlier commit and fails now, the search for a repair is constrained to whatever changed between that commit and the present, rather than spanning the entire manuscript. `git bisect`, applied to a build failure exactly as it would be applied to a software regression, automates this search: given a known-good commit and a known-bad one, it checks out successive commits between them, compiling at each, until it identifies the exact commit that introduced the failure, at which point the diff for that single commit is almost always small enough to make the actual cause immediately visible.

Small, logically coherent commits, as Chapter 4 already recommended for reasons of readable history, pay an additional dividend here: a bisection that lands on a commit touching a single chapter file is far more informative than one landing on a commit that happened to touch a dozen unrelated files at once, since the former narrows the search to a specific paragraph while the latter still leaves a meaningful amount of investigation to do even after the offending commit is identified. Compilation, treated this way, stops being a final hurdle checked only when a chapter feels finished and becomes a continuous verification process, ideally run on every commit through the CI setup from Chapter 15, so that a failure is caught and localized to a single small commit immediately rather than discovered much later, once dozens of subsequent changes have accumulated on top of it and the search space has grown accordingly.

## 19.6 Isolating Package and Workflow Changes

The same discipline extends to package upgrades and workflow changes, not only to content edits. Introducing a new document class, switching bibliography systems, or upgrading a graphics package should be done in its own isolated commit, ideally its own isolated test build, so that any resulting incompatibility has an unambiguous origin. Changing several such things simultaneously, upgrading a package and switching engines and revising the class file all in one pass, makes diagnosis considerably harder than it needs to be, since a resulting failure could plausibly trace to any of the changes, and confirming which one is actually responsible requires undoing them one at a time regardless of whether they were

introduced one at a time or all at once. Introducing them one at a time in the first place avoids that reconstruction entirely.

Debugging in LaTeX, treated with this discipline throughout, is ultimately an exercise in preserving continuity. A project has a most recent working state, and the purpose of debugging is to find the smallest repair that reconnects the current document to that state while keeping whatever genuine improvement motivated the change that broke it. This mirrors good software engineering generally: a reliable project is maintained not through heroic rewrites undertaken in a moment of frustration with a broken build, but through careful observation of what the compiler is actually reporting, incremental modification narrow enough to be individually verifiable, and disciplined repair aimed at the smallest change that restores a working document.

Part VII closes here. Part VIII turns from individual documents to the corpus-level discipline of writing and maintaining book-length works over years, and to the broader ecosystem of tools a superuser's daily workflow depends on.

# Part VIII

# Professional Workflows

# Chapter 20

# Writing Book-Length Works

Writing a book of several hundred pages is not simply a matter of extending the techniques used for a short article. At a certain scale, questions of architecture become more important than questions of typography. The challenge shifts from typesetting individual pages to managing an evolving system whose components must remain coherent over months or years of development. The superuser therefore thinks less about individual documents and more about maintaining a corpus whose structure can continue to grow without becoming unmanageable.

## 20.1 Structuring a 500-Page Monograph

The first architectural decision concerns the hierarchy of the document itself. Chapters are the natural unit of organization for most books because they correspond to complete conceptual arguments. A chapter should be readable as a coherent piece of writing even when considered independently of the surrounding text. Parts exist at a higher level. They communicate to the reader that several chapters together contribute to a broader theme and that a significant conceptual transition occurs between groups of chapters. Introducing Parts simply because a book is long adds visual complexity without improving comprehension. A monograph should therefore employ the Part level only when the reader benefits from understanding that the book consists of several major intellectual phases rather than a continuous sequence of chapters.

This same principle applies to the source tree. Chapter 3 argued that modularity improves maintainability only while the resulting modules remain meaningful units of work. At book scale this observation becomes even more important. Dividing a manuscript into chapter files is almost always beneficial. Individual chapters compile cleanly, navigation remains straightforward, version-control history is easy to interpret, and conceptual revisions naturally occur at chapter boundaries. Breaking every section or subsection into its own file, however, usually crosses the point of diminishing returns. The source tree becomes fragmented, readers of the source spend more time following chains of `\input` directives than reading content, and editing a single argument may require opening half a dozen tiny files. Excessive modularity recreates the same cognitive fragmentation that software engineers encounter when every function occupies its own source file.

The opposite mistake is equally common. Allowing a single chapter file to accumulate hundreds of pages eventually recreates the original single-file manuscript from which modularity was intended to escape. Navigation becomes increasingly difficult, merge conflicts become more frequent, and unrelated revisions become entangled within the same source. The appropriate level of modularity therefore lies between these extremes. Each file should represent a conceptually complete component whose size remains comfortable for editing and review. When the boundaries of those components cease to correspond to meaningful intellectual units, the architecture should be reconsidered.

Large revisions are inevitable. As a research programme matures, arguments become clearer, new chapters emerge, and previously independent discussions may naturally merge. Conversely, a chapter that originally seemed unified may eventually divide into two distinct subjects. Such restructuring should be viewed as an architectural refactoring rather than a failure of planning. The organization of the manuscript ought to evolve alongside the understanding it expresses.

## 20.2 Consistency of Notation and Terminology Across a Corpus

Single books demand internal consistency; a corpus demands historical consistency. Readers who move from one volume to another should encounter familiar notation, familiar terminology, and familiar structural conventions unless there is a compelling reason for change. Achieving this requires more than a shared macro file. It requires maintaining an explicit editorial style guide that records preferred terminology, notation, capitalization conventions, theorem styles, citation practices, and naming decisions. Such a document becomes the authoritative reference for the entire corpus rather than relying upon memory or habit.

Shared notation files, common document classes, and centralized bibliographic resources naturally extend into this editorial layer. Decisions made once should rarely be revisited without careful consideration, since every alteration introduces the possibility of inconsistency across previously published work. When a symbol or technical term genuinely requires refinement, the change should be treated as an evolution rather than a silent replacement. A reader encountering both the earlier and later works should be able to understand that the terminology has narrowed, broadened, or shifted in meaning, and why that change occurred.

Periodic terminology audits provide the publishing analogue of software maintenance. Just as bibliographic consistency benefits from occasional review, technical vocabulary benefits from systematic inspection across the entire corpus. Symbols should continue to denote the same concepts, recurring definitions should retain consistent wording where appropriate, and synonymous expressions introduced during drafting should either be unified or intentionally distinguished. Such reviews reduce unnecessary variation while preserving genuine conceptual development.

Consistency does not imply rigidity. New ideas occasionally require new notation, and improved understanding sometimes reveals weaknesses in earlier terminology. The objective is therefore not to freeze the language permanently but to ensure that change itself remains documented, intentional, and comprehensible.

## 20.3 Front Matter, Back Matter, and Reader Expectations of a "Real" Book

The front matter of a book performs an orienting function distinct from the main text. It establishes the relationship between the reader and the work before the first chapter begins. Within a corpus, this role expands further. A preface may introduce readers to the broader research programme while simultaneously explaining the particular contribution of the current volume. Achieving both aims without making the book appear dependent upon earlier titles requires careful editorial discipline. Each book should remain self-contained while still acknowledging its place within a larger body of work.

Consistency across the series reinforces this sense of coherence. Colophons, edition notices, publication metadata, acknowledgements, and explanatory notes concerning the relationship between volumes should appear in predictable locations and follow common conventions. Readers gradually learn where to locate such information, and the books acquire a recognizable editorial identity extending beyond typography alone.

Back matter deserves equal attention. Appendices, bibliographies, indexes, glossaries, and notes should reflect consistent organizational principles throughout the corpus rather than varying arbitrarily between titles. The result is a collection of books that behaves as a unified publishing project rather than as unrelated manuscripts sharing only an author's name.

Revision becomes particularly demanding once a manuscript reaches book scale. Entire chapters may migrate between Parts, arguments may be reorganized into a more coherent progression, and extensive renumbering may follow naturally from those structural improvements. Such revisions are best approached through the same version-control discipline introduced earlier in the book. Large reorganizations should consist of small, understandable changes whose history preserves not only the final arrangement but the reasoning that led to it. The architecture of a successful monograph is therefore never static. Like the software systems discussed throughout this book, it remains maintainable because every revision respects the underlying structure rather than discarding it wholesale.

The next chapter surveys the toolchain this discipline actually runs on: the compiler-adjacent tools, package management, and the criteria for choosing what to depend on when a project is expected to remain buildable for years.

# Chapter 21

# The TeX Ecosystem

LaTeX itself occupies only one layer of a modern publishing workflow. Around the compiler exists an ecosystem of supporting tools responsible for bibliography management, indexing, glossary generation, syntax highlighting, build orchestration, package installation, and environment management. Beginners often encounter these utilities individually, adding them only when a particular feature requires them. The superuser instead regards them as components of a single publishing infrastructure whose behaviour should be understood, documented, and reproduced as carefully as the source itself. A document is therefore not merely a collection of `.tex` files, but a complete computational environment capable of regenerating the publication from first principles.

## 21.1 The Compiler-Adjacent Tools as a Coherent Set

Earlier chapters introduced tools such as `latexmk`, `arara`, `biber`, `makeindex`, `glossaries`, and `minted` as they became necessary within the publishing process. At book scale, however, they cease to be isolated utilities and instead form a coordinated toolchain. Each contributes one stage to the transformation from source files to a finished publication, and each should therefore be configured as part of the project rather than as an incidental property of an individual workstation.

Build orchestration tools such as `latexmk` and `arara` determine how compilation proceeds. They encode the dependency graph of the project, ensuring that bibliography generation, index creation, glossary processing, and repeated compilation occur in the correct order without requiring manual intervention. Rather than remembering a sequence of compiler invocations, the project itself records the build procedure. This transforms compilation from a collection of commands into a reproducible specification.

Bibliography processors and indexing utilities occupy the next layer. Their outputs are not independent artefacts but intermediate representations consumed by subsequent compilation passes. They therefore belong within the documented build pipeline rather than being treated as optional post-processing steps. A project whose bibliography only builds correctly on the author's machine has failed to capture an essential component of its own construction.

The `minted` package illustrates another important principle. Unlike most LaTeX pack-

ages, `minted` depends upon software external to the TeX ecosystem, namely the Pygments syntax highlighter. Successful compilation therefore requires more than a correctly installed TeX distribution. It requires the presence of compatible Python software and a functioning external execution environment. The lesson extends beyond `minted` itself. Every external dependency becomes part of the publishing system and must therefore be documented, versioned, and considered during long-term maintenance. Reproducibility extends beyond TeX Live alone to encompass the complete computational environment.

## 21.2 Package Management and Cross-Machine Reproducibility

Package management is often regarded as routine maintenance. For long-lived publishing projects it instead becomes an editorial concern. Updating every installed package immediately upon release may provide access to new functionality, but it also introduces unnecessary uncertainty into documents intended to remain buildable for many years. A project whose output changes unpredictably following unrelated package updates cannot reasonably be described as reproducible.

The distinction between TeX Live and MiKTeX matters less than maintaining a documented, consistent environment throughout the life of a project. Whatever distribution is chosen should become part of the project's specification. Version numbers, compiler engines, package revisions, and auxiliary tools should be recorded alongside the source so that future builds occur under substantially the same conditions as earlier ones.

Utilities such as `tlmgr` provide the practical means to manage this stability. Rather than treating package updates as continuous background activity, the superuser upgrades deliberately, verifies that existing projects continue to compile correctly, and records significant environmental changes within version control. The build environment becomes another component of the project subject to the same discipline as the manuscript itself.

The strongest guarantee of reproducibility comes from containerized builds. A container image containing a known TeX Live release, together with every external dependency required by the project, transforms environmental assumptions into executable specifications. Instead of relying upon written instructions describing how a system should be configured, the configuration itself becomes part of the reproducible artefact. Future compilation therefore depends less upon reconstructing an historical workstation and more upon executing a preserved environment.

For most long-term projects, remarkably few tools are actually necessary. A carefully selected compiler, a build manager, a bibliography processor, an indexing utility, a syntax-highlighting system where required, and a stable package distribution are sufficient for the overwhelming majority of technical books. Standardizing upon this modest collection across an entire corpus simplifies maintenance, reduces unexpected interactions, and allows improvements to propagate consistently between projects.

## 21.3 Choosing Packages for Longevity, Not Novelty

The richness of CTAN encourages experimentation, but large publishing projects reward conservatism. Every package introduced into a manuscript becomes another long-term dependency whose continued maintenance directly affects the buildability of the document. Selecting packages therefore resembles selecting software libraries for production systems rather than experimenting with isolated examples.

Several characteristics distinguish mature packages from transient ones. Widely adopted packages are typically incorporated into standard distributions, maintained over many years, documented thoroughly, and used as dependencies by numerous other packages. Their interfaces evolve cautiously because changes affect a substantial portion of the LaTeX ecosystem. Such packages acquire stability through sustained community use rather than temporary popularity.

Conversely, some packages exhibit warning signs from the outset. They may depend upon undocumented implementation details of a particular engine, receive little ongoing maintenance, or derive their popularity primarily from recent online tutorials rather than long-term adoption. These characteristics do not necessarily imply poor quality, but they do increase the risk that future projects will inherit unnecessary maintenance burdens.

The consequences of choosing poorly often emerge only after considerable investment. Once a package becomes deeply integrated into a large manuscript, replacing it requires revisiting macros, document structure, and accumulated assumptions spread throughout hundreds of pages. Alternatively, the abandoned package may need to be incorporated directly into the project and maintained locally, transferring responsibility for its continued operation from the original author to the publisher of the manuscript. Neither outcome is attractive.

For this reason, experienced LaTeX users tend toward deliberate conservatism. Novel packages should justify their inclusion through substantial, enduring benefit rather than incremental convenience. A mature, well-supported solution that remains functional for decades is generally preferable to a fashionable alternative whose long-term viability remains uncertain. The objective is not to minimize innovation but to maximize the lifespan of the publishing system itself. When viewed over the lifetime of a corpus rather than a single document, stability is itself a valuable feature.

The final chapter draws these threads together, situating LaTeX itself as one stage of a larger continuous workflow that begins with editing and ends with publication.

# Chapter 22

# LaTeX as One Stage of a Larger Pipeline

The central argument of this book has been that LaTeX is best understood not as an isolated typesetting language but as one component of a larger publishing system. Throughout the preceding chapters we have examined document architecture, macro programming, typography, bibliographic management, graphics generation, automation, and reproducible builds. Each contributes to a single objective: transforming structured knowledge into a durable publication. That objective cannot be achieved by LaTeX alone. Every finished document originates from an editing environment, often depends upon external computation, and ultimately passes through a reproducible publishing pipeline. Understanding those surrounding stages completes the superuser's view of document production.

## 22.1 Editing: Vim as the Input Layer

Every publishing system begins with editing. Source files are created, revised, reorganized, and reviewed long before the compiler is invoked. For the superuser, the editor is therefore not merely a place to enter text but the environment in which the structure of the project is constructed.

A modal editor such as Vim complements LaTeX particularly well because both systems assume that structured text deserves structured manipulation. Multi-file navigation, rapid movement between chapters, powerful search and substitution facilities, macros for repetitive editing tasks, and programmable commands allow large manuscripts to remain manageable without requiring the author to leave the keyboard. A chapter ceases to be a static document and instead becomes another source file within a navigable project tree.

The correspondence extends further. LaTeX encourages semantic markup rather than visual formatting, while modal editing encourages operations upon syntactic structure rather than individual characters. The combination naturally supports large technical manuscripts in which consistency, repeatability, and navigation matter more than interactive page layout. The editing environment therefore serves as the input layer of the publishing pipeline, producing structured source suitable for subsequent computation rather than polished presentation.

## 22.2 Building: Rust as the Computation Layer

Modern technical publications frequently contain material that cannot reasonably be maintained by hand. Figures originate from numerical simulations, tables summarize experimental results, benchmarks evolve as software changes, and diagrams may themselves be generated programmatically. Between editing and publishing therefore lies computation.

Earlier chapters presented graphics pipelines, automated table generation, and build-time processing as practical techniques. Here they become part of a broader architectural view. Programs written in Rust, Python, Julia, R, or similar languages consume structured input, perform computation, and generate intermediate artefacts destined for inclusion within the document. The publication itself becomes the final presentation layer of a computational process rather than an independent description of that process.

For computational tasks upon which the correctness of the document depends, reliability becomes an editorial concern rather than merely a programming concern. Languages that encourage explicit error handling, deterministic behaviour, and strong static guarantees reduce the likelihood that incorrect results propagate silently into published work. The computational layer therefore deserves the same engineering discipline applied throughout the rest of the publishing system.

## 22.3 Publishing: LaTeX as the Output Layer

Only after editing and computation have completed does LaTeX assume responsibility for publication. At this stage, structured text, generated figures, bibliographic data, glossaries, indexes, and computed results are assembled into a coherent document whose typography communicates the underlying ideas clearly and consistently.

Earlier chapters argued that the source tree rather than the PDF should be regarded as the primary artefact of a publishing project. That principle now acquires its full significance. LaTeX occupies the final transformation within a sequence of representations. Editing produces structured source. Computation produces validated results. Publication integrates both into a finished work intended for readers rather than authors or software systems.

The boundaries between these stages should remain explicit. Editors manipulate ideas expressed as structured text. Computational programs manipulate data and produce derived artefacts. LaTeX manipulates presentation, pagination, references, typography, and layout. When each stage remains responsible for its own concerns, the entire workflow becomes easier to understand, maintain, and extend.

## 22.4 One Continuous Workflow

The superuser therefore experiences publishing not as a sequence of unrelated tasks but as a single continuous pipeline. A change introduced during editing propagates naturally through computation into publication without requiring manual intervention. Version control records every modification, build systems coordinate every dependency, continuous integration verifies every revision, and the publishing environment regenerates every affected output automatically.

Consider a simulation whose numerical parameters change following a theoretical refinement. The revised source is committed to version control. The build system recompiles the computational program, producing updated datasets. Those datasets regenerate figures and tables consumed by the LaTeX manuscript. The document recompiles, cross references stabilize, bibliographies regenerate where necessary, and a new PDF emerges whose contents reflect the revised computation. At no stage is information copied manually between programs. Every transformation remains documented, reproducible, and recoverable through the project's recorded history.

It is worth tracing this example through the specific mechanics established earlier in the book, since the continuity being described is not an abstraction but a concrete chain of already-introduced tools. The parameter change lands in a single, logically scoped commit, per the discipline of Chapter 4. A `make figures` target, of the kind described in Chapter 5 and extended in Chapters 14 and 15, invokes the Rust program, writes its output as CSV alongside any computed values destined for inline macros, and a dependent `make book` target then runs `latexmk`, which in turn triggers PGFPlots and `pgfplotstable` to read the regenerated CSV directly, exactly as Chapter 14 described. Nothing in this chain requires a human to notice that a number has changed and manually update a table; the table is wrong only until the next build, and correct again the moment it runs. Wired into continuous integration as in Chapter 15 and Chapter 18, the same sequence fires on every push, so a parameter change that quietly breaks a downstream figure, rather than merely updating it, is caught within minutes rather than discovered by a reader months later. This is the concrete shape "the pipeline" takes once every earlier chapter's individual technique is allowed to compose with the others rather than being applied in isolation.

This continuity distinguishes a publishing pipeline from an editing workflow. The former describes the complete movement of information from initial conception to finished publication. Each stage accepts structured input, performs a well-defined transformation, and produces output suitable for the next stage. Because every transformation is recorded, the entire process remains reproducible long after the original work has been completed.

## 22.5 Closing Thesis

The opening chapter argued that LaTeX should not be understood as a sophisticated word processor. The chapters that followed have attempted to justify that claim from progressively broader perspectives. Document architecture and version control demonstrated that large publications benefit from software-engineering principles. Macro programming showed that documents are programmable systems rather than static text. Typography illustrated that presentation itself can be specified precisely rather than adjusted interactively. Bibliographic management established the importance of traceability and consistency. Graphics, automation, and computation transformed figures and tables from manually maintained artefacts into reproducible products of documented pipelines. Document engineering extended these principles to projects spanning multiple books, editions, and years of development.

This final chapter places those ideas within a single framework. Editing provides structured input. Computation derives validated results. LaTeX publishes both through a programmable, reproducible typesetting system. None of these stages is sufficient in isolation,

yet together they form an environment capable of producing technical documents whose content, presentation, and history remain transparent, maintainable, and repeatable.

The superuser therefore learns far more than a collection of commands or packages. They learn to design publishing systems. Once that perspective has been adopted, LaTeX ceases to be the destination of the workflow and instead becomes one carefully engineered stage within a larger process of transforming ideas into enduring, professionally published knowledge.

## 22.6 Beyond This Book

Several threads in the preceding chapters lead further than this book itself goes, and are worth naming directly rather than leaving implicit. The appendices exist for the reader who wants the internals behind a technique rather than only the technique: TeX's own primitive layer beneath the macros this book spent most of its time on, and just enough category theory and type theory to make precise the handful of asides, in Chapters 3, 6, 7, and 11, that leaned on that language. Neither appendix is required to use anything covered in the main chapters; both exist for the reader who finds that the "why" behind a working technique is itself worth understanding.

The editing and computation layers introduced in this chapter are, correspondingly, each the subject of their own dedicated treatment elsewhere: what it means to think in a modal editor rather than merely operate one, and what it means to write computation with the same reliability discipline this book has asked of a publishing pipeline. Read together with those companion volumes, the three describe a single continuous practice, input, computation, and publication, rather than three unrelated skills that happen to be useful in adjacent contexts. This book has tried to earn its place as the last of the three: not the most visible stage of the work, but the one responsible for making everything that came before it legible, durable, and worth having written down.

# Appendix A

# TeX Internals for Programmers

Chapter 1 introduced boxes, glue, penalties, tokens, and category codes at the level needed to start writing and debugging LaTeX documents productively. This appendix goes further, into the actual mechanics underneath that introduction, for the reader who wants to understand not just that a particular behavior happens but exactly why, down to the primitive level. It is written for a reader comfortable with programming language internals generally, tokenizers, parsers, evaluators, and treats TeX as what it is: a small, unusually literal, and unusually stable programming language whose only unusual feature is that its typical output is a typeset page rather than a return value.

Nothing in the main chapters requires this appendix. It exists for Chapters 1, 6, 8, and 19 specifically, for a reader who wants the full mechanism behind the model those chapters use.

## A.1 The Category Code Table in Full

Every character TeX reads is assigned one of sixteen category codes at the moment it is read, determined by a table that can be modified at any point during processing. The categories are: 0, escape character, marking the start of a control sequence; 1, begin group, ordinarily {; 2, end group, ordinarily }; 3, math shift, ordinarily `$`; 4, alignment tab, ordinarily `&`; 5, end of line; 6, parameter, ordinarily `#`; 7, superscript, ordinarily `^`; 8, subscript, ordinarily `_`; 9, ignored; 10, space; 11, letter; 12, other; 13, active, a character that behaves as a control sequence in its own right; 14, comment character, ordinarily `%`; 15, invalid character; and a small number of characters TeX treats specially at the input-line level before category codes are even consulted.

Two consequences follow directly from this table's mutability. First, `\catcode` assignments are what packages like `verbatim` actually rely on: temporarily reassigning most characters to category 12 (other) suspends their special meaning entirely, which is why text inside a `verbatim` environment can contain unescaped \, {, }, and `%` without any of them being interpreted. Second, active characters, category 13, are the mechanism behind packages that make a character like `~` or `"` behave as a macro invocation, `babel`'s language-specific shortcuts and `csquotes`'s smart quote handling both work by making an otherwise ordinary character active and defining what it expands to.

## A.2 Tokenization

TeX's input processor reads one line at a time, applies the category code table current at that moment to convert the line's characters into tokens, and hands the resulting token list to the expansion and execution machinery described in the following sections. A token is either a control sequence, formed by an escape character followed by a run of letter-category characters (or a single non-letter character), or a character token, a pairing of a character and its category code.

This is worth stating precisely because it clarifies a common point of confusion: `\ab` and `\a` followed by `b` are not automatically the same two-token or one-token sequence; whether `\ab` tokenizes as the single control sequence `\ab` or as `\a` followed by a literal `b` depends entirely on whether `b` has category 11 (letter) at the moment that line is read, since control sequence names absorb a run of consecutive letter-category characters. This is also why an explicit space or a non-letter character is needed after certain control sequences to prevent TeX from trying to absorb further letters into the control sequence name that was not intended, and why `\ ` (control sequence followed by an explicit space) is TeX's idiom for "a control sequence name ends here, and additionally insert a space," distinct from simply writing the next character with nothing between.

## A.3 Expansion in Detail

Chapter 6 introduced expansion as TeX's core mechanism: an expandable token is replaced by what it expands to, repeatedly, until nothing further expandable remains at the position currently under consideration. The detail worth adding here is exactly which tokens count as expandable, since this determines what `\edef`, `\expandafter`, and the various `\if` conditionals can and cannot reach.

Expandable tokens include macros (anything defined with `\def` or built on top of it), the conditional primitives (`\ifnum`, `\ifdim`, `\ifx`, and the rest of the `\if` family), `\csname...\endcsname` for constructing a control sequence name from its component characters, `\expandafter` itself, and a handful of others such as `\input` (which expands to the contents of the named file) and `\the` (which expands to the printable representation of a register's current value). Non-expandable tokens include ordinary character tokens once they have reached this stage, box-construction and assignment primitives, and anything that has a side effect on TeX's internal state rather than producing further tokens.

This distinction is what makes `\expandafter` meaningful rather than merely a curiosity: `\expandafter\A\B` expands `\B` once (if it is expandable) before `\A` is examined at all, letting a macro writer reach past one token to affect what comes immediately after it, which is the standard technique for constructing a control sequence name out of pieces via `\csname` or for testing the category or value of whatever comes next in the input before committing to how the current macro should proceed. Chained `\expandafter` calls, `\expandafter\expandafter\expandafter\A\expandafter\B\C`, extend the same idea to reach further into an input stream than a single application would allow, at the cost of a syntax that is notoriously difficult to read without deliberate practice; the `expl3` function-based interface exists largely to make constructions like this unnecessary in ordinary macro-writing.

## A.4 Grouping and Scope

{ and }, category codes 1 and 2, delimit a group, and a group is TeX's only mechanism for scoping an assignment. Any assignment made inside a group, a font change, a counter change, a redefinition of an existing macro via local `\def`, is automatically undone at the group's closing brace, restored to whatever value it held before the group opened. This is why `{\bfseries bold text} normal text` correctly returns to the surrounding font without any explicit "turn off bold" command: the closing brace itself performs the restoration, because the font change was a local assignment scoped to that group.

`\global` overrides this behavior for a specific assignment, making it persist past the end of the current group rather than being undone; `\def` versus `\gdef`, and the various registers' local versus global assignment forms, follow the same pattern throughout TeX. Environments (`\begin{...}...\end{...}`) are, at the primitive level, built from exactly this grouping mechanism, `\begin{env}` opens a group and `\end{env}` closes it, which is why an assignment made inside an environment is automatically local to it without any special environment-specific scoping machinery being required.

## A.5 Registers: TeX's Only Variables

TeX has no general-purpose variables in the sense a conventional programming language would; it has a fixed set of typed register classes, each with a fixed number of numbered slots, and LaTeX's counters, lengths, and named boxes are all built on top of these registers rather than being a separate mechanism. Count registers hold integers and back LaTeX's `\newcounter` and every `\value`-accessible counter. Dimen registers hold a length with an attached unit and back `\newlength`. Skip registers hold glue, a dimension plus stretch and shrink components, and back the vertical and horizontal spacing lengths a document defines. Box registers hold a constructed box, and are what `\newsavebox` and `\sbox`/`\savebox` allocate and populate. Toks registers hold a token list, unexpanded, and are used less often directly by document authors but underlie some of the more advanced macro-programming techniques in Part III.

`\newcount`, `\newdimen`, and their relatives, which LaTeX's `\newcounter` and `\newlength` are thin wrappers around, allocate a previously unused numbered register and bind a control sequence name to it, which is why TeX historically imposed a hard limit on the total number of registers of each type available at once (256 in classic implementations, considerably larger in modern engines including LuaTeX), and why careless repeated allocation of registers inside a macro that gets called many times, rather than allocating once at load time, was historically a real resource concern, though less pressing in current engines than it was in the era the limit was originally set.

## A.6 Dimensions and Glue

A TeX dimension is a number followed by a unit: `pt` (point), `in`, `cm`, `mm`, `em` and `ex` (relative to the current font's size), among others, all ultimately converted internally to a common fixed-point representation in units of scaled points. Arithmetic on dimensions, adding two

lengths, multiplying a length by an integer, is supported directly by TeX's register and assignment syntax, though it is limited compared to a general-purpose numeric type; `calc` and, in the LaTeX3 kernel, the expandable arithmetic in `l3fp`, extend this considerably for macro-writing purposes.

Glue, introduced conceptually in Chapter 1, is formally a dimension with two additional components: a stretch amount and a shrink amount, each itself a dimension, optionally given in "fil" units (`fil`, `fill`, `filll`) representing infinite stretch or shrink of increasing order, used for constructions like `\hfill` that should absorb all remaining space on a line regardless of how much space that turns out to be. When TeX justifies a line or a page, it computes the total natural width of the material on that line or page, compares it to the target width, and distributes the required stretch or shrink proportionally across all the glue present, weighted by each glue item's stretch or shrink component, which is the actual mechanism behind the paragraph justification described at a higher level in Chapter 1.

## A.7 Penalties and the Line-Breaking Algorithm

TeX's paragraph-breaking algorithm, generally credited to Knuth and Plass [2], does not choose line breaks greedily, filling each line as full as possible and moving to the next; it considers the entire paragraph at once, evaluating the total "badness" of every combination of break points and choosing the set of breaks that minimizes an accumulated cost function across the whole paragraph, subject to the penalties attached to each candidate break point. A break with an attached penalty near the forcing threshold ($-10000$ or beyond) is effectively mandatory; a large positive penalty makes a break progressively less likely to be chosen even where the surrounding glue could accommodate it; penalties near zero leave the decision almost entirely to the badness calculation.

This global, whole-paragraph optimization is why TeX's line breaking is generally regarded as superior to a simpler greedy algorithm: a greedy algorithm can leave one line very loose and the next disproportionately tight without any way to trade slack between them, while Knuth-Plass considers the paragraph as a whole and can accept slightly more stretch on one line in exchange for a much better break somewhere else in the same paragraph. The page-breaking algorithm applies a related but distinct badness-and-penalty evaluation to the vertical dimension, considering where to break across pages given the accumulated material and any explicit penalties (widow and orphan control, `\nopagebreak`, and so on) that discourage or force a page break at a specific point.

## A.8 Boxes and Box Construction

A box, as introduced in Chapter 1, is either an `\hbox` (horizontal box, built by placing its contents side by side) or a `\vbox` (vertical box, built by stacking its contents), each with a width, height, and depth (the depth being the extent below the baseline, relevant for characters like `g` or `y` that descend below it). `\vtop` is a variant of vertical box construction whose reference point is its first line rather than its last, useful when aligning the tops rather than the bottoms of adjacent material.

Every piece of typeset material is ultimately assembled into boxes recursively: a character becomes a small box supplied by the current font, a word is an `\hbox` of character boxes with interword glue, a line is an `\hbox` of word boxes, a paragraph is a `\vbox` of line boxes, and a page is, at the top level, a `\vbox` containing everything on it. `\raise`, `\lower`, `\moveleft`, and `\moveright` shift an already-constructed box relative to its surrounding context without changing its own internal structure, and are the primitives underlying LaTeX-level positioning commands used for fine typographic adjustment.

## A.9 The Page Builder and Output Routine

Material accumulates in what TeX calls the "main vertical list" as a document is processed, and the page builder periodically checks whether enough material has accumulated to fill a page, using the same badness-and-penalty evaluation described in A.7 applied to page-length material rather than paragraph-length material. When it decides a page break should occur, it invokes the output routine, a token list, customizable via `\output`, responsible for taking the box representing everything accumulated for that page (available inside the output routine as `\box255`) and actually placing it, adding headers, footers, and page numbers, and sending the result to the DVI or PDF driver.

This is the mechanism packages that modify page layout, custom headers and footers, multi-column output, watermarking, actually hook into: `fancyhdr` and similar packages redefine `\output` or the hooks it calls, rather than modifying anything about how individual pages are composed at a lower level. Understanding that headers and footers are inserted by the output routine, at the point a page is finalized, rather than being part of each page's content from the start, explains why certain header/footer customizations interact awkwardly with certain float placements or multi-column layouts: both are competing for control over material that has already been provisionally assigned to a specific page by the page builder, and the output routine is the single point where that assignment can still be revised.

## A.10 A First Script in LuaTeX

LuaTeX, the engine underlying LuaLaTeX, embeds a full Lua interpreter and exposes hooks into nearly every stage of TeX's processing described in this appendix: the input processor, the tokenizer, the paragraph and page builders, and the output routine can all be observed or modified from Lua code, in addition to Lua being usable simply as a general-purpose scripting language callable from within a document via `\directlua`.

A minimal first use of this capability, connecting directly to the build-time computation described in Chapters 14 and 15, is reading an external value into the document from Lua rather than through LaTeX's own, more limited macro-programming facilities: `\directlua{tex.sprint(2^10)}` evaluates a Lua expression and inserts its printed result directly into the document, which for anything beyond simple arithmetic, reading a JSON or CSV file's contents directly rather than through a dedicated LaTeX package, calling out to an external library available to Lua, becomes a genuinely useful capability rather than a curiosity. This is, deliberately, as far as this appendix goes into LuaTeX scripting; a reader

who wants to build substantially on this capability is better served by the `luatex-plain` and LuaTeX-specific documentation than by an appendix whose purpose is to explain the primitives underneath ordinary LaTeX use, not to teach Lua programming.

With this appendix's model of tokens, expansion, grouping, registers, boxes, and the page builder in hand, the behavior described throughout Chapters 1 through 19, why a macro expands the way it does, why a fragile command breaks under reprocessing, why a debugging session traces expansion rather than execution, should read as the direct, predictable consequence of a small number of rules applied consistently, rather than as a collection of separately memorized special cases.

# Appendix B

# A Category Theory Primer

This appendix is scoped narrowly. It exists to make two things in the main text precise for a reader who wants that precision: the commutative-diagram notation introduced practically in Chapter 11, and the gluing intuition offered as an aside in Chapter 3. It is not a general introduction to category theory, and nothing in the main chapters requires having read it. A reader who is comfortable simply drawing a `tikz-cd` diagram and trusting that "the two paths agree" is the right way to read it can skip this appendix entirely without losing anything Chapters 1 through 22 depend on.

## B.1 Objects and Morphisms

A category consists of a collection of objects and, between certain pairs of objects, arrows called morphisms. A morphism from an object $A$ to an object $B$, written $f : A \to B$, should be thought of as a structure-respecting way of getting from $A$ to $B$, though what "structure-respecting" means depends entirely on which category is under discussion. This is deliberately abstract, and the abstraction is the point: the same small set of rules, described below, applies whether the objects are sets and the morphisms are functions, or the objects are TeX's layout primitives from Chapter 1 and the morphisms are ways of combining them.

Concretely, for the layout model from Chapter 1: take boxes as objects. A morphism from one box to another can be thought of as a way of placing the first box so that it becomes part of the second, a horizontal concatenation, a vertical stack, a nesting. This is not a rigorous formalization of TeX's box model, and this appendix does not attempt one; it is offered only as a concrete anchor, so that "object" and "morphism" mean something a LaTeX-familiar reader can picture rather than remaining purely abstract from the outset.

## B.2 Composition and Identity

Two morphisms $f : A \to B$ and $g : B \to C$ compose into a single morphism $g \circ f : A \to C$, read "$g$ after $f$": first travel the arrow $f$, then the arrow $g$. Composition is required to be associative, $(h \circ g) \circ f = h \circ (g \circ f)$, meaning that when three morphisms are chained together, it does not matter whether the first two are composed before the third is added or

the last two are composed before the first is added; the result is the same morphism either way.

Every object $A$ has an identity morphism $\mathrm{id}_A : A \to A$, which does nothing: composing it with any morphism leaves that morphism unchanged, $f \circ \mathrm{id}_A = f$ and $\mathrm{id}_B \circ f = f$ for $f : A \to B$. This is the categorical analogue of a no-op, and it exists in essentially every concrete category one could construct, boxes placed nowhere relative to themselves, a function returning its own input unchanged, a macro that expands to exactly its own argument.

These two properties, associative composition and an identity for every object, are the entire definition of a category. Everything else in this appendix is built from applying that definition to specific, useful examples.

## B.3 Reading a Commutative Diagram

A diagram, in this sense, is a picture of some objects and some of the morphisms between them. A diagram commutes if every pair of paths between the same two objects, following the arrows in the direction they point, compose to the same overall morphism. This is the entire content of the word "commutative" in "commutative diagram," and it is worth stating plainly because the word sounds more mysterious than the idea actually is: it just means every route from here to there gets you to the same place, in the same way.

Chapter 11 introduced the `tikz-cd` syntax for a two-by-two square:

```
\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "h"] \\
C \arrow[r, "k"'] & D
\end{tikzcd}
```

This diagram asserts, by being drawn as a commutative square (the convention, unless stated otherwise, in most mathematical writing that uses this notation), that going from $A$ to $D$ by first applying $f$ then $h$ gives the same result as going from $A$ to $D$ by first applying $g$ then $k$: $h \circ f = k \circ g$. A diagram with more objects and arrows asserts the same thing for every pair of paths it contains between any two objects appearing in it, not merely the outer boundary. This is why diagrams are useful for the kind of argument Chapter 11 mentioned, diagram chasing, tracing an element or a relationship through several routes and confirming, or using, the fact that they must agree: the diagram itself is a compact assertion of exactly which routes are claimed to agree, and a proof "by diagram chase" is typically an argument that walks through the diagram's commutativity to derive something not immediately obvious from any single arrow alone.

## B.4 Functors

A functor is a structure-preserving map from one category to another: it sends each object of the source category to an object of the target category, and each morphism to a morphism, in a way that respects composition and identity, $F(g \circ f) = F(g) \circ F(f)$ and $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$. Informally, a functor is a way of translating an entire category, objects and the relationships

between them, into another category without breaking any of those relationships in the process.

Chapter 10's discussion of custom notation and font loading offers a concrete, LaTeX-relevant instance of this idea, stated there without the formal vocabulary: take a category whose objects are the semantic tokens a document's notation distinguishes (a variable, an operator, a boundary), and whose morphisms capture how those tokens can combine or relate to each other in the document's own notational conventions. A font, together with the macros mapping notation to glyphs, can be read as a functor from that category into a category of rendered marks on a page, sending each semantic token to the glyph or glyph-combination that represents it, in a way that preserves the relationships between tokens (an operator applied to two variables renders as those two variable-glyphs combined via the operator's glyph, not as something unrelated to either). This is offered as an interpretation, not a proof; the value of stating it this way is that it makes precise something Chapter 10 already argued informally, that changing a document's notation system, in the sense of swapping which functor is being used, is a different and more tractable kind of change than rewriting the document's actual mathematical content, because the underlying category of semantic tokens and their relationships can stay fixed while only the target category (the specific glyphs and rendering) changes.

## B.5 The Gluing Intuition, Stated More Precisely

Chapter 3 offered an aside, kept deliberately informal in the main text, that a book-length document's chapters are pieces that must agree with each other wherever they touch, and that a broken cross-reference is a failure of exactly that agreement. The precise version of this intuition belongs to sheaf theory, a further layer of structure built on top of category theory, and is worth sketching briefly here for the reader who wants to see where the informal statement actually comes from, without this appendix attempting a full treatment of sheaves, which would go considerably beyond what the main text needs.

Informally: imagine covering a document by its constituent chapter files, each an open piece of the whole. Assign to each piece the data it actually contains, its labels, its definitions, its notation. A presheaf is simply this assignment, piece by piece, together with a consistent way of restricting a larger piece's data down to a smaller piece contained within it (the whole book's accumulated labels, restricted to just what Chapter 5 itself defines). A presheaf satisfies the sheaf condition, becomes a sheaf, when data that agrees on every overlap between pieces can always be glued into a single, consistent piece of data on the union of those pieces; the gluing axiom is precisely the requirement that local agreement is sufficient for global consistency, with no further obstruction. A broken `\ref`, in this language, is exactly a failure of that gluing condition: some label's data, as recorded by the chapter that defines it, disagrees with (or is entirely missing from) what a different chapter's reference to it expects, and LaTeX's two-pass compilation, described mechanically in Chapter 5 and Appendix A, is the concrete procedure by which the document's build process checks, and where possible enforces, that this gluing condition actually holds before producing final output.

This is offered, as Chapter 3 itself notes, as an intuition rather than a formal claim the book depends on; nothing about actually fixing a broken cross-reference requires un-

derstanding sheaves, and the practical fix, described in Chapter 3, is simply to find where two pieces of the document have drifted out of agreement and reconcile them. The formal language is included here only because, for a reader who already thinks in these terms, it names precisely what "agreement" and "gluing" mean in a way that "the pieces have to match" leaves only informal.

## B.6 Where to Go Further

This appendix has covered exactly enough to read a `tikz-cd` diagram with genuine understanding of what commutativity asserts, and to see precisely what the informal gluing language in Chapter 3 is gesturing at. For a reader who wants to go further into category theory itself, independent of anything this book needs, a standard first text such as Riehl's *Category Theory in Context* [5] or Awodey's *Category Theory* [6] covers the material here in full rigor and continues considerably further, into limits, adjunctions, and the broader machinery that a single appendix scoped to two chapters' worth of asides was never trying to replace.

# Appendix C

# A Type Theory Primer

Like Appendix B, this appendix is scoped narrowly, to exactly what makes precise a handful of asides in the main text: the `xparse` argument specifiers from Chapter 6, the key-value interface design discussed in Chapter 7, and the document-class-as-type-constructor framing from Chapter 16. It is not an introduction to type theory as a foundation for mathematics or to the dependently typed proof assistants (Coq, Agda, Lean) that a reader might encounter elsewhere under this name. Nothing in the main chapters requires having read this appendix; a reader content to treat an `xparse` specifier as "the pattern of arguments this macro accepts" already has everything Chapters 6, 7, and 16 need.

## C.1 Types as Descriptions of Shape

A type, in the narrow sense this appendix needs, is a description of the shape a piece of data is allowed to take: what it can be, before asking what its actual value is on any given occasion. Saying "this argument has type integer" says nothing about which integer; it says only that whatever value shows up there, it will be one of the things an integer can be, not a string, not a list, not something malformed. This is a useful description to attach to a macro's arguments for exactly the reason it is useful in an ordinary programming language: it lets an error be caught, or at least a mismatch be recognized, based on shape alone, before anyone has to reason about the specific value involved.

TeX itself has no type system in any of these senses; a macro argument is, at the primitive level, an unexamined sequence of tokens, and it is entirely the macro's own body that decides what to do with it, including whether to check its shape at all. Everything in this appendix is therefore describing a discipline layered on top of TeX by tools like `xparse` and `l3keys`, and by an author's own conventions, not a property TeX itself enforces the way a statically typed compiler would.

## C.2 Simple Types and Macro Signatures

Chapter 6 introduced `xparse`'s argument specifiers, `m` for mandatory, `o` for optional, `s` for an optional star, `d` for a custom-delimited argument, combined into a specifier string describing a macro's full calling convention. Read through the lens of this appendix, a specifier string

like `mo` is a simple type for the macro's argument list: it says the macro accepts exactly two arguments, the first always present, the second present or absent, and nothing about what values those arguments can meaningfully hold once supplied, only their presence, absence, and delimiting syntax.

This is deliberately a weak type system, in the sense that it checks shape (how many arguments, delimited how) without checking content (whether the first argument, once captured, is actually a sensible thing for the macro's body to do with). A macro declared with specifier `m` still receives whatever token sequence the caller wrote for that argument, with no guarantee it makes sense as, say, a length or a color name; any deeper checking has to be written into the macro's body explicitly, using the conditionals from Chapter 7 or an explicit format check. The value of the `xparse` specifier, even at this shallow level, is that it makes a macro's calling convention legible at the point of definition, `\NewDocumentCommand{\keyword}{m o}{...}` states the shape directly, rather than requiring a reader to infer the same information by reading through a `\def`-based implementation that manually tests for and strips an optional bracket.

## C.3 Records and Key-Value Interfaces

A key-value interface, covered practically in Chapter 7, corresponds to what a type theorist would call a record type: a structure with several independently named fields, each with its own type, where a value of the record type provides (or, for optional fields, may omit) a value for each field. `\examplebox[boxed=false]{...}`, the worked example from Chapter 7, is supplying a partial record: a value for the `boxed` field, and, implicitly, defaults for `title` and `numbered`, the two fields left unspecified.

This framing clarifies something about good key-value interface design that Chapter 7 argued for on more informal grounds: every field of a well-designed record type has a sensible default, or is a genuinely required field the type does not permit omitting, and a record type's fields are, absent some explicit dependency between them, independently specifiable, which is exactly the composability property Chapter 7 asked a good key-value interface to have. A key-value interface that violates composability, where setting one key silently changes what another key means, is, in this language, not really a record type at all but something closer to a variant or tagged union pretending to be a record, and is harder to reason about, and to document, for exactly the reason a variant type disguised as a record would be harder to reason about in a conventional statically typed language.

## C.4 Dependent Records: When One Key's Validity Depends on Another

Some key-value interfaces genuinely do need one key's valid values, or its very presence, to depend on another key's value: a `format` key that is only meaningful once a `type` key has been set to a specific value, or a `numbered` key whose valid range depends on which numbering scheme a separate `scheme` key selected. This is what a dependent record type describes: a record where a later field's type is itself computed from an earlier field's value, rather than being fixed in advance independent of everything else in the record.

`l3keys` supports this pattern directly, though not by that name, through key groups and conditional key definitions, where a key's own definition can inspect the current value of another key already processed earlier in the same call and behave differently depending on it. Recognizing this as "the interface has a genuine dependency between two of its fields" rather than merely "this key's behavior is a little complicated" is useful design guidance in its own right: a dependent key-value interface is harder to document, harder to validate, and harder for a caller to use correctly than an interface whose fields are all independent, so a dependency worth having is one earning its complexity, not one introduced merely because the implementation happened to make it easy to check one key's value while processing another.

## C.5 Document Classes as Type Constructors

Chapter 16 described a custom document class as something that can require and structure a document's front-matter fields, title, author, affiliation, at the class level, raising an error at compile time if a required field is missing. In this appendix's terms, a document class defined this way is acting as a type constructor: `\documentclass{myresearchclass}` does not merely load a set of macros, it instantiates a structured type, the class's own notion of "a valid document of this class," with required fields that must be supplied (title, author) and optional fields with defaults (a subtitle, defaulting to none).

A class-level check that a required field is missing is, in this framing, a type error caught at compile time rather than a mistake a human proofreader has to catch by inspection, exactly the property Chapter 16 argued made this worth building into the class rather than leaving to convention. This is also why Chapter 16's advice to pin a shared class's version per book, so that an earlier book's build does not change when the shared class evolves, matters in the same way that changing a type's definition in a conventional codebase requires care: every existing "value" of that type, every book already built against an earlier version of the class, is implicitly relying on the type's earlier shape, and a change to what the type requires or permits can silently invalidate documents that were previously well-formed under the old definition.

## C.6 Where This Appendix Stops

This appendix has covered exactly enough to read Chapter 6's `xparse` specifiers as lightweight signatures, Chapter 7's key-value interfaces as record types (dependent, where the interface genuinely needs it), and Chapter 16's document classes as type constructors enforcing a required shape at compile time. It has deliberately not covered the machinery a reader might associate with "type theory" from a more formal context: the Curry-Howard correspondence between types and propositions, dependent function types in the fuller sense used by proof assistants, or anything resembling a soundness proof for the informal type discipline described here, since TeX's macro system was never designed to support one and nothing in this book's practical guidance depends on one existing. A reader who wants that fuller treatment is better served by a dedicated type theory text, such as Pierce's *Types and Programming Languages* [7] for the programming-languages perspective this appendix has

been borrowing vocabulary from, than by an appendix whose entire purpose was to stay exactly as large as three chapters' worth of asides required.

# Appendix D

# Glossary of Notation and Terms

This glossary was compiled by scanning the manuscript's own recurring terms and notation rather than written independently of it, in keeping with the recommendation that it be produced early enough to catch drift before the editorial pass. Each entry points back to the chapter or chapters where the term is introduced or most fully discussed.

## D.1 A Note on the Consistency Check

A pass across all twenty-two chapters and Appendix A found the manuscript's core vocabulary consistent: engine names (`LuaLaTeX`, `XeLaTeX`, `pdfTeX`), package names, and the book's recurring concepts (fragile/robust commands, expansion versus execution, the modularity threshold, the shared-resource pattern applied to notation/bibliography/style) are used the same way everywhere they recur. Two points are worth noting, neither of which needs correction:

`pdfLaTeX` and `pdfTeX` are used deliberately as distinct terms, `pdfTeX` for the underlying engine, `pdfLaTeX` for LaTeX running on it, and this distinction is intentional rather than an inconsistency.

Chapter 7's discussion of key-value interfaces covers `pgfkeys` and `l3keys` specifically rather than `xparse`, which is introduced instead in Chapter 6 for argument specification. The two topics are adjacent but distinct, and the manuscript's actual chapter contents are more precise on this point than the original outline's phrasing suggested; no change is needed, but a reader cross-checking this glossary against the outline in the table of contents should trust the chapters themselves.

## D.2 Glossary

**admissible / inadmissible state** — A document state that does (or does not) compile to valid, intended output. Used in Chapter 19 to frame debugging as finding the minimal repair back to an admissible state, and referenced in Chapter 13 as a worked example term for index and glossary construction.

**arara** — A build-automation tool driven by directives written as comments inside the `.tex` file itself, offered in Chapter 5 as an alternative to `latexmk` for projects needing per-file

build variation, and revisited in Chapter 21 as part of the coordinated toolchain.

**biber / BibTeX / BibLaTeX** — `BibTeX` is the original external bibliography processor, reading a `.bib` database and a style file to produce formatted citations. `Biber`, paired with the `BibLaTeX` package, is its more capable modern successor. Chapter 12 frames the choice between this pairing and `thebibliography` as a portability-versus-scale threshold.

**box (`\hbox`, `\vbox`, `\vtop`)** — A rectangular region of fixed width, height, and depth; TeX's primitive unit of typeset material. Introduced in Chapter 1, treated in full in Appendix A.9.

**category code** — The classification (one of sixteen categories) TeX assigns to each character at the moment it is read, determining whether it behaves as a letter, a control-sequence marker, a comment character, and so on. Introduced in Chapter 1, given in full in Appendix A.1.

**corpus (shared resource pattern)** — This book's term for a body of related books or essays sharing notation, bibliography, and style conventions. The recurring pattern, introduced with the shared notation file in Chapter 10 and extended to bibliographies (Ch. 12), nomenclature (Ch. 13), document classes (Ch. 16), and an explicit editorial style guide (Ch. 20), is to maintain such resources once, centrally, rather than reinvent them per document.

**expansion vs. execution** — The distinction between a token being rewritten into further tokens (expansion) versus a primitive being carried out with a side effect (execution). Central to Chapters 1, 2, and 6; given in full in Appendix A.3.

**fontspec** — The `LuaLaTeX`/`XeLaTeX` interface for loading OpenType and TrueType fonts directly, including their feature sets (ligatures, small capitals, old-style figures). Covered in Chapter 9 and extended to custom and multilingual fonts in Chapter 10.

**fragile / robust command** — A command is fragile if it does not survive being written to an auxiliary file and reprocessed (as section titles and captions are); it is robust if it does. Introduced in Chapter 2 as a place LaTeX's abstraction leaks, addressed directly in Chapter 8.

**glue** — A flexible space with a natural width plus stretch and shrink components, used throughout TeX for justification and vertical spacing. Introduced in Chapter 1, given in full in Appendix A.6.

**key-value interface** — An argument-passing convention (`option=value, option=value`) letting a macro's settings be named explicitly rather than positional. Covered in Chapter 7 via `pgfkeys` and `l3keys`, designed with the same discipline as a small API: defaults, validation, composability.

`l3keys` / `pgfkeys` — Two implementations of the key-value pattern; `pgfkeys` originates with PGF/TikZ, `l3keys` is the LaTeX3 kernel's own version. Chapter 7.

**latexmk** — A build-automation tool that inspects a project's dependencies and runs the correct sequence of compiler and auxiliary-tool passes automatically, including a continuous "preview" mode for drafting. Chapter 5; revisited as part of the coordinated toolchain in Chapter 21.

**macro as rewrite rule (macro as term)** — This book's framing of a LaTeX macro not as a procedure call but as a rewrite rule operating on token streams, following directly from TeX's expansion model. Introduced in Chapter 1, developed through Chapters 2 and 6.

**minted** — A syntax-highlighting package for code listings, notable for depending on the external Pygments tool rather than being self-contained within the TeX ecosystem. Chapter 15's use for verified listings; Chapter 21's use as the example of an external, non-TeX dependency that still needs documenting for reproducibility.

**modularity threshold** — The point at which splitting a project into more files stops improving maintainability and starts fragmenting it. Introduced in Chapter 3 at the level of an individual project, revisited in Chapter 20 at the scale of a full monograph.

**namespacing (macro prefixing)** — The convention of prefixing project-specific macro names (`\rsvpfield` rather than `\field`) to avoid collisions with package-defined names. Chapter 8.

**nomenclature** — A listing of symbol meanings, distinct from a glossary of terms, generated from the same shared notation file a project already maintains rather than kept as a separate list. Chapter 13.

**notation system (private/shared)** — A deliberately designed, consistently reused set of symbols and operators for a recurring technical framework, maintained as a single shared file across a corpus rather than invented per document. Chapter 10, extended by Chapter 11's discussion of theorem styles and delimiters and Chapter 20's terminology audits.

**output routine** — The token list, customizable via `\output`, responsible for finalizing each page (headers, footers, page numbers) once the page builder has decided where a page break occurs. Appendix A.9; the mechanism behind packages like `fancyhdr`.

**penalty** — A signed integer attached to a candidate line or page break, discouraging, encouraging, or forcing a break there. Introduced in Chapter 1, given in full alongside the Knuth-Plass line-breaking algorithm in Appendix A.7.

**publishing pipeline (source tree as primary artifact)** — This book's framing, developed across Chapters 14, 15, and 18, of a project's repository, not any single rendered PDF, as the actual authoritative artifact, with every output (book, website, excerpt) a derived rendering of it.

**reproducibility** — The property that a project, checked out at any point in its history, can be rebuilt into the output it originally produced, by anyone, given a documented environment. Introduced in Chapter 4, extended to package management and containerized builds in Chapter 21.

**superuser** — This book's term, used throughout and reflected in its title, for a reader who already knows ordinary LaTeX usage and is extending that knowledge into project architecture, macro design, and long-term maintenance.

`tikz-cd` — A `TikZ`-based package for typesetting commutative diagrams, treated in Chapter 11 as a directly useful practical tool independent of the book's occasional category-theoretic framing, and given a short conceptual grounding in Appendix B.

`\tracingmacros / \show` — Diagnostic primitives for observing macro expansion directly, `\tracingmacros` for a full trace, `\show` for a single targeted query. Chapter 19.

`unicode-math` — The `fontspec`-layered package for loading a dedicated OpenType math font, keeping text and mathematics typographically consistent. Chapter 9.

`xparse` — The LaTeX3-kernel package providing compact argument specifiers (`m`, `o`, `s`, `d`, and others) for defining a macro's calling convention explicitly. Chapter 6.

# Bibliography


[1] Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1984.

[2] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981.

[3] Leslie Lamport. *LaTeX: A Document Preparation System*, 2nd edition. Addison-Wesley, 1994.

[4] Frank Mittelbach and Michel Goossens. *The LaTeX Companion*, 2nd edition. Addison-Wesley, 2004.

[5] Emily Riehl. *Category Theory in Context*. Dover Publications, 2016.

[6] Steve Awodey. *Category Theory*, 2nd edition. Oxford University Press, 2010.

[7] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[8] Till Tantau. *The TikZ and PGF Packages: Manual*. CTAN, 2013.

[9] Will Robertson and Khaled Hosny. *fontspec: XeLaTeX and LuaLaTeX font selection*. CTAN, 2022.

[10] The LaTeX3 Project. *xparse: Generic document command parser*. CTAN, 2023.

[11] Philip Lehman, Philipp Kime, Audrey Boruvka, and Joseph Wright. *The biblatex Package*. CTAN, 2023.

[12] Nicola Talbot. *glossaries.sty: Create glossaries and lists of acronyms*. CTAN, 2022.

[13] Konrad Rudolph and Geoffrey M. Poore. *The minted package: Highlighted source code in LaTeX*. CTAN, 2022.

[14] Hans Hagen, Taco Hoekwater, and Luigi Scarso. *LuaTeX Reference Manual*. CTAN, 2016.