

The `comment2tex` package

Erik Nijenhuis

2026/07/22 v1.1

Abstract

`comment2tex` typesets a source file that carries its documentation in special comments: the comments become ordinary $\text{\LaTeX}$, the rest becomes a listing. It ships a Lua converter and two $\text{\TeX}$ wrappers — one for $\text{\LaTeX}$ /Lua $\text{\LaTeX}$ and one for plain $\text{\TeX}$ — both providing `\includebash`, `\includelua`, `\includeyaml` and `\includemake`.

Contents

1	Introduction	2
1.1	Is it literate programming?	2
2	Usage	3
2.1	Examples in this manual	3
3	Engines and use cases	4
4	The build	4
4.1	Configuration	5
4.2	Names and the distribution	5
4.3	Targets	5
5	The converter, by example	7
5.1	The converter	7
5.1.1	Styles and wrappers	7
5.1.2	Option handling	8
5.1.3	Conversion	9
5.1.4	Tangling	10
5.1.5	File helpers	10
5.1.6	The $\text{\TeX}$ -facing entry point	11
5.1.7	Command-line interface	11
6	The listings variant	13
6.1	Styling	14
7	Implementation	14
7.1	The $\text{\LaTeX}$ wrapper	14
7.2	The plain $\text{\TeX}$ wrapper	17

8	Testing	19
8.1	What the suite covers	20
8.2	Harness helpers	20
8.3	Fixtures	21
8.4	Driver documents	22
8.5	The converter as a command-line program	24
8.6	The L ^A T _E X wrapper	25
8.7	The plain T _E X wrapper	26
8.8	The documentation itself	26
8.9	Summary	26
9	CI/CD	27
9.1	The build workflow	27
9.2	The publish workflow	28

1 Introduction

A source file may document itself: lines beginning with a chosen *doc-comment* prefix are prose, everything else is code. For Bash, YAML and Make the prefix is a double hash (`##`, itself a comment in all three); for Lua it is a triple dash (`---`). The converter `comment2tex.lua` turns such a file into L^AT_EX: doc lines are emitted verbatim (minus the prefix) and runs of code are wrapped in a listing.

This package is the T_EX front end. `\includebash`, `\includelua`, `\includeyaml` and `\includemake` take a source file, run it through the converter, and input the result, so a document can pull in annotated sources without a manual build step.

1.1 Is it literate programming?

Loosely. `comment2tex` borrows literate programming’s vocabulary — it *weaves* an annotated source into typeset documentation and can *tangle* it back to runnable code — but it is a deliberately lightweight take, not an implementation of Knuth’s WEB.

Two things set it apart. First, order: classic literate programming writes the source for the reader — code is split into named chunks presented in whatever sequence best explains it, and *tangle* reassembles them into the order the machine needs. `comment2tex` does no reordering; the source *is* the program, in its natural order. Second, *tangle* is off the critical path: in WEB you must tangle a `.web` before you can compile or run it, whereas here the doc-comments are ordinary language comments, so the annotated file runs unmodified and *tangling* — just dropping the doc lines — is an optional convenience, never a build step. The prose follows the code, not the other way round.

That fits its origin. `comment2tex` was written to document Xerdi’s production-critical sources: Bash scripts, and the Ansible playbooks and inventories woven into a complete infrastructure handbook. Such sources are mostly linear and functional, so WEB’s reordering would buy little — and since they are deployed and run as-is, needing no tangle step between the documented file and the running one was a hard requirement, not a nicety. Each stays a short, standalone file, documented in place and pulled into the book by the include macros rather than

dissolved into one monolithic source, so the file on disk stays the single source of truth.

What is left is the low-friction core — one source that reads as documentation and runs as code, with no separate web language and no chance of the two drifting apart. In spirit it sits beside a self-documenting `.dtx` (this manual is one) rather than WEB or CWEB.

2 Usage

`\includebash` `\includebash{<file>}` typesets a Bash source (doc prefix `##`); `\includelua{<file>}` `\includelua` a Lua source (doc prefix `---`); `\includeyaml{<file>}` a YAML source and `\includeyaml` `\includemake{<file>}` a Makefile (both doc prefix `##`). All four derive an output name by replacing the file’s extension with `.c2t.tex`, generate that file with the appropriate *style* and *wrapper*, and `\input` it. The `<file>` carries at most one extension; an extensionless name (such as a `Makefile`) is fine and keeps its whole self as the base.

Because `##` is an ordinary YAML comment, the double-hash doc lines stay valid YAML: the annotated file remains loadable as-is, so `\includeyaml` typesets the very file that ships — no stripped copy, no risk of drift.

`\ctxuselistings`
`\ctxuseverbatim`

Verbatim or listings. Under L^AT_EX the include macros render in one of two modes, switchable anywhere in the document. The default is a built-in *numbered verbatim* — no dependency, no setup, the same plain look as this manual’s own listings. `\ctxuselistings` switches subsequent includes to `listings` output (syntax highlighting, line breaking, the full `\lstset` vocabulary); `\ctxuseverbatim` switches back. Both obey T_EX grouping, so a single block can opt in locally. Requesting `listings` without having loaded it warns once and stays on verbatim. Like l^AT_EX’s own `macrocode`, the verbatim renderer does not break long lines; switch to `listings` mode (*The listings variant*, with `breaklines`) where wrapping matters.

In `listings` mode the package applies a plain default `\lstset` for you (`\AtBeginDocument`, only if `listings` is loaded). The Bash, Lua and Make styles name languages `listings` ships itself (`make` is loaded on demand), so they need no help. Only `yaml` is missing from `listings`; `\includeyaml` handles that gracefully by installing an empty `yaml` language if the document has not defined its own (so `language=yaml` resolves to the default style rather than faulting). It does *not* do the same for `Make`: since `make` exists but loads lazily, pre-installing an empty one would shadow the real highlighter. The plain T_EX wrapper always uses a verbatim listing, since `listings` is L^AT_EX-only.

2.1 Examples in this manual

This manual is built with the very macros it documents, so each one appears as a full worked example on a real file:

- `\includemake` on a `Makefile` — see *The build*.
- `\includelua` on the converter itself — see *The converter, by example*.
- `\includebash` on a short script, both verbatim and through `listings` — see *The listings variant*.

- `\includebash` on the cross-engine test suite — see *Testing*.
- `\includeyaml` on the GitHub Actions workflows — see *CI/CD*.

Together they cover all four styles and both render modes. Even the code in *Implementation* is typeset automatically — there by `ltxdoc`’s `macrocode` rather than `comment2tex` — so no listing in this manual is typed by hand.

3 Engines and use cases

How the conversion happens depends on the engine.

Lua^ATeX (recommended). LuaTeX embeds Lua, so the wrapper loads `comment2tex.lua` as a library and converts *in process* through `\directlua`. No shell escape, no separate run: just `luaAtex document`.

pdf^ATeX with shell escape. pdfTeX cannot run Lua, so with `--shell-escape` the wrapper calls `texlua comment2tex.lua` through `\write18` to generate the fragment, then inputs it. Run `pdflatex -shell-escape document`.

pdf^ATeX, separate run (no shell escape). Generate the fragments ahead of time and let a normal run merely `\input` them. For each source run `texlua comment2tex.lua --style <style> -o <base>.c2t.tex <file>` (or `make`), then `pdflatex document`. If a required fragment is missing the package raises an error telling you which command to run.

plain TeX. `comment2tex.tex` provides the same two macros for plain TeX. Under `luatex` it converts in process; under `pdfAtex`/`tex` it uses the same shell-escape or separate-run fallback.

Engine	In process	Otherwise
Lua ^A TeX	yes (<code>\directlua</code>)	—
pdf ^A TeX	no	<code>-shell-escape</code> or separate run
plain <code>luatex</code>	yes	—
plain <code>pdf^Atex</code> / <code>tex</code>	no	<code>-shell-escape</code> or separate run

In every case the generated file is `<base>.c2t.tex`, so the in-process, shell-escape and separate-run routes are interchangeable.

4 The build

The package’s own build is a `Makefile`, and — like the converter and the test suite — it is an annotated source, typeset here by `\includemake` (the Make style, doc prefix `##`). It drives the converter under `texlua`, builds this manual, and assembles the CTAN upload; its `##` lines are the prose you read, the rest is the Make code that runs.

Like the test suite, the `Makefile` is a development artefact and is not part of the distribution, so the listing appears only when it is present.

```
1 # comment2tex --- Makefile
```

4.1 Configuration

The programs the build calls, and a watch switch. The converter is invoked as `$(C2T)` throughout; `make doc WATCH=1` keeps `latexmk` running (`--pvc`) so the manual rebuilds on every save.

```
2 TEXLUA   = texlua
3 TEX      = tex
4 LATEXMK  = latexmk
5 C2T      = $(TEXLUA) comment2tex.lua
6
7 WATCH ?=
8 PVC      = $(if $(WATCH),--pvc,)
9
```

4.2 Names and the distribution

The documented source and its docstrip driver, the generated wrappers, the converter, the manual, and the test suite.

```
10 NAME     = comment2tex
11 DTX      = comment2tex.dtx
12 INS      = comment2tex.ins
13 WRAPPERS = comment2tex.sty comment2tex.tex
14 DOC      = comment2tex.pdf
15 README   = README.md
16 TESTSH   = comment2tex-test.sh
17 WORKFLOWS = $(wildcard .github/workflows/*.yml)
18
```

`FRAGS` names the standalone `.c2t.tex` fragments a source weaves to; the converter documents itself with the `---` Lua style.

```
19 LUA_SRC  = comment2tex.lua
20 FRAGS    = $(LUA_SRC:.lua=.c2t.tex)
21
```

`DISTFILES` are shipped to CTAN: the hand-written sources plus the README and the built PDF. Per CTAN's upload guide no generated or derived file goes in — `comment2tex.sty/.tex` are regenerated from the `.ins`, and `comment2tex.lua` ships as its annotated source (its `---` lines are ordinary Lua comments, so it runs unmodified) rather than a tangled copy. The test suite is a development artefact and is not shipped either; the manual's *Testing* section `\includebash`s it only when present (guarded by `\IfFileExists`), so the bundled sources still build the PDF without it. See the README's CTAN submission section for the full rationale.

```
22 DISTFILES = $(DTX) $(INS) $(README) $(LUA_SRC) $(DOC)
23
24 .PHONY: all wrappers doc test frags package clean help
25
26 all: doc
27
```

4.3 Targets

`wrappers` extracts the runtime `.sty/.tex` from the `.dtx` with `docstrip`.

```
28 wrappers: $(WRAPPERS)
29 $(WRAPPERS): $(DTX) $(INS)
```

```

30 $(TEX) $(INS)
31
    doc typesets the manual; latexmk reruns LuaLATEX until the cross-references
    settle. It depends on the test suite too, so the Testing section restages whenever
    the harness changes.
32 doc: $(DOC)
33 $(DOC): $(DTX) comment2tex.sty comment2tex.lua $(TESTSH) $(WORKFLOWS)
34 $(LATEXMK) $(PVC) --lualatex --interaction=nonstopmode $(DTX)
35
    test runs the cross-engine suite (comment2tex-test.sh).
36 test:
37     ./comment2tex-test.sh
38
    frags weaves the literate sources into standalone .c2t.tex fragments; the
    pattern rule uses the Lua style for a .lua source.
39 frags: $(FRAGS)
40
41 %.c2t.tex: %.lua comment2tex.lua
42     $(C2T) --style lua -o $@ $<
43
    package assembles the CTAN zip: only DISTFILES, under a single comment2tex/
    directory, with no generated or derived file.
44 package: $(NAME).zip
45 $(NAME).zip: $(DISTFILES)
46     $(RM) -r $(NAME) $(NAME).zip
47     mkdir $(NAME)
48     cp $(DISTFILES) $(NAME)/
49     zip -r $(NAME).zip $(NAME)
50     $(RM) -r $(NAME)
51
    clean removes every generated file — the wrappers, fragments, the PDF and
    the TEX auxiliaries.
52 clean:
53     $(RM) -r $(NAME) $(NAME).zip
54     $(RM) $(WRAPPERS) $(FRAGS) $(DOC) *.c2t.tex \
55         comment2tex.aux comment2tex.toc comment2tex.log \
56         comment2tex.idx comment2tex.ilg comment2tex.ind comment2tex.glo \
57         comment2tex.out comment2tex.hd comment2tex.fls comment2tex.fdb_latexmk
58
    help lists the targets (echoed literally, so make help needs no extra tooling).
59 help:
60     @echo "Targets:"
61     @echo "  all          build the documentation (default)"
62     @echo "  doc          build comment2tex.pdf (WATCH=1 keeps rebuilding)"
63     @echo "  wrappers    extract comment2tex.sty and comment2tex.tex"
64     @echo "  frags       convert literate sources to .c2t.tex fragments"
65     @echo "  package     build a CTAN-ready zip (comment2tex.zip)"
66     @echo "  test        run the cross-engine test suite"
67     @echo "  clean       remove generated files"

```

5 The converter, by example

Because the converter is itself a literate Lua file, the cleanest demonstration is to let it document itself. The remainder of this section is produced by `\includelua{comment2tex.lua}` — the generated documentation demonstrates the converter's functionality under Lua_{La}T_EX. It renders through the default verbatim path (numbered, dependency-free), the same look as the code listings elsewhere in this manual.

```
1 #!/usr/bin/env texlua
```

5.1 The converter

This file is the engine behind `\includebash` and `\includelua`. It plays two roles from one source: a command-line program run under `texlua`, and a Lua library loaded *in process* by the T_EX wrappers under Lua_{La}T_EX.

Run as a program it reads a source file and emits L_AT_EX on stdout (or to `-o`). Loaded as a library — `require"comment2tex"` or `loadfile(...)("comment2tex")` — it returns a module table whose `write` entry converts a file directly, so Lua_{La}T_EX needs neither shell escape nor a separate run.

The two modes are told apart by the first vararg of the chunk: `require` (and our explicit loader) pass the module name `"comment2tex"`, whereas `texlua` passes the command-line arguments. Per the Lua_{La}T_EX manual, `texlua` is a stand-alone Lua 5.3 interpreter that fills the global `arg` table just like stock `lua`.

```
2 local LIBRARY_MODE = (... == "comment2tex")
3
4 local M = {}
5
```

5.1.1 Styles and wrappers

A *style* pairs a doc-comment prefix with a listing language; a *wrapper* is the pair of `begin/end` lines emitted around a code block. Both are presets so that a single `--style/--wrapper` selects sensible defaults, while `--comment`, `--language`, `--begin` and `--end` still override any individual field.

```
6 M.styles = {
7   bash = { comment = "##", language = "bash" },
8   lua  = { comment = "---", language = "[5.3]Lua" },
9   yaml = { comment = "##", language = "yaml" },
10  make = { comment = "##", language = "make" },
11 }
12
```

The `lstlisting` wrapper targets L_AT_EX (the `listings` package); the `plain` wrapper targets plain T_EX, where `\ctxlisting` reads its body verbatim up to a line equal to `\endctxlisting`. Templates substitute `@LANG@` with the language and `@CONT@` with `firstnumber=last`, on every block after the first (empty on the first) so numbering stays continuous. A wrapper may also carry a `preamble`, emitted once before the first block; the `plain` wrapper uses it to announce the code-line count (as `@WIDTH@`, a run of zeros matching its digit width) so the reader can size the line-number gutter.

```
13 M.wrappers = {
14   lstlisting = {
```

```

15     begin = "\\begin{lstlisting}[language=@LANG@,@CONT@numbers=left]",
16     finish = "\\end{lstlisting}",
17 },
18 plain = {
19     preamble = "\\ctxlistingtotal{@WIDTH@}",
20     begin = "\\ctxlisting%",
21     finish = "\\endctxlisting",
22 },
23 }
24
25 M.defaults = {
26     style = "bash",
27     wrapper = "lstlisting",
28     comment = nil,
29     language = nil,
30     begin = nil,
31     finish = nil,
32     preamble = nil,
33 }
34

```

5.1.2 Option handling

`new_opts` layers explicit overrides on top of the defaults; `resolve` then fills any still-empty field from the chosen style and wrapper presets.

```

35 function M.new_opts(over)
36     local o = {}
37     for k, v in pairs(M.defaults) do o[k] = v end
38     if over then
39         for k, v in pairs(over) do
40             if v ~= nil then o[k] = v end
41         end
42     end
43     return o
44 end
45
46 function M.resolve(o)
47     local style = M.styles[o.style]
48     if not style then
49         error("comment2tex: unknown style: " .. tostring(o.style)
50             .. " (expected bash, lua, yaml or make)")
51     end
52     local wrapper = M.wrappers[o.wrapper]
53     if not wrapper then
54         error("comment2tex: unknown wrapper: " .. tostring(o.wrapper)
55             .. " (expected lstlisting or plain)")
56     end
57     o.comment = o.comment or style.comment
58     o.language = o.language or style.language
59     o.begin = o.begin or wrapper.begin
60     o.finish = o.finish or wrapper.finish
61     o.preamble = o.preamble or wrapper.preamble
62     return o
63 end

```

5.1.3 Conversion

`expand_tabs` turns tabs into spaces to the next `M.tabwidth` stop, so a listing shows clean indentation instead of a raw tab (which a monospace font would typeset as a missing-glyph box). It is applied only when *weaving* code lines; `\tangle` leaves the source byte-for-byte, so a `Makefile`'s recipe tabs survive.

```

65 M.tabwidth = 4
66
67 local function expand_tabs(line, w)
68   if not line:find("\t", 1, true) then return line end
69   local out, col = {}, 0
70   for i = 1, #line do
71     local c = line:sub(i, i)
72     if c == "\t" then
73       local n = w - (col % w)
74       out[#out + 1] = string.rep(" ", n)
75       col = col + n
76     else
77       out[#out + 1] = c
78       col = col + 1
79     end
80   end
81   return table.concat(out)
82 end
83

```

The source is walked line by line. A line that starts with the comment prefix is documentation: any open code block is closed and the line is emitted with the prefix (and one optional following space) stripped. Anything else is code: a block is opened if one is not already running and the line is emitted with tabs expanded. A final close guarantees the listing is shut even when the file ends inside code.

```

84 function M.convert(o, lines, emit)
85   local prefix = o.comment
86   local plen = #prefix
87   local in_code = false
88   local block_count = 0
89
90   if o.preamble then
91     local n = 0
92     for _, line in ipairs(lines) do
93       if line:sub(1, plen) ~= prefix then n = n + 1 end
94     end
95     emit((o.preamble:gsub("@WIDTH@", ("0"):rep(#tostring(n))))
96   end
97
98   local function open_code()
99     if not in_code then
100       block_count = block_count + 1
101       local cont = block_count == 1 and "" or "firstnumber=last,"
102       local line = o.begin:gsub("@LANG@", o.language):gsub("@CONT@", cont)
103       emit(line)
104       in_code = true

```

```

105     end
106 end
107
108 local function close_code()
109     if in_code then
110         emit(o.finish)
111         in_code = false
112     end
113 end
114
115 for _, line in ipairs(lines) do
116     if line:sub(1, plen) == prefix then
117         close_code()
118         emit((line:sub(plen + 1):gsub("^ ", "")))
119     else
120         open_code()
121         emit(expand_tabs(line, M.tabwidth))
122     end
123 end
124 close_code()
125 end
126

```

5.1.4 Tangling

The inverse of weaving: `tangle` drops every doc-comment line and emits the rest of the source unchanged, so an annotated file yields its runnable form without a separate `sed/grep` pass. The lines it keeps are exactly the lines `convert` would have numbered, so a tangled file and its typeset listing agree on line numbers.

```

127 function M.tangle(o, lines, emit)
128     local prefix = o.comment
129     local plen = #prefix
130     for _, line in ipairs(lines) do
131         if line:sub(1, plen) ~= prefix then
132             emit(line)
133         end
134     end
135 end
136

```

5.1.5 File helpers

`read_lines` slurps a file and splits it into lines, keeping a final unterminated line if present. `convert_file` returns the converted `LATEX` as a string; `write_file` sends it to outfile.

```

137 function M.read_lines(path)
138     local fh, err = io.open(path, "r")
139     if not fh then
140         error("comment2tex: cannot open input: " .. tostring(err))
141     end
142     local data = fh:read("*a")
143     fh:close()
144     local lines = {}

```

```

145 for line in (data .. "\n"):gmatch("(.-)\n") do
146     lines[#lines + 1] = line
147 end
148 if data:sub(-1) == "\n" then
149     lines[#lines] = nil
150 end
151 return lines
152 end
153
154 function M.convert_file(path, o)
155     o = M.resolve(o or M.new_opts())
156     local out = {}
157     M.convert(o, M.read_lines(path), function(line) out[#out + 1] = line end)
158     return table.concat(out, "\n") .. "\n"
159 end
160
161 function M.tangle_file(path, o)
162     o = M.resolve(o or M.new_opts())
163     local out = {}
164     M.tangle(o, M.read_lines(path), function(line) out[#out + 1] = line end)
165     return table.concat(out, "\n") .. "\n"
166 end
167
168 function M.write_file(infile, outfile, o)
169     local text = M.convert_file(infile, o)
170     local fh, err = io.open(outfile, "w")
171     if not fh then
172         error("comment2tex: cannot open output: " .. tostring(err))
173     end
174     fh:write(text)
175     fh:close()
176     return outfile
177 end
178

```

5.1.6 The $\text{\TeX}$ -facing entry point

`\includebash` and `\includelua` call this through `\directlua`: it converts `infile` to `outfile` for the given style and wrapper, entirely in process. Keeping the signature positional keeps the $\text{\TeX}$ side trivial.

```

179 function M.write(style, wrapper, infile, outfile)
180     return M.write_file(infile, outfile,
181         M.new_opts{ style = style, wrapper = wrapper })
182 end
183

```

5.1.7 Command-line interface

Parsing mirrors the documented options; `die` reports to `stderr` and exits non-zero. Only reached when the file is executed by `texlua`, never when it is loaded as a library.

```

184 local function usage(stream)
185     stream:write([[
186 Usage: comment2tex.lua [options] <input>

```

```

187
188 Weave a source with embedded LaTeX doc-comments into LaTeX, or with
189 --tangle strip the doc-comments back to the runnable source.
190
191 Options:
192 -s, --style NAME          bash, yaml, make (##) or lua (---) [default: bash]
193 -w, --wrapper NAME        lstlisting or plain [default: lstlisting]
194 -c, --comment PREFIX      doc-comment prefix marking a doc line
195 -l, --language LANG       listing language for code blocks
196 -b, --begin TEMPLATE      listing begin template (@LANG@, @CONT@)
197 -e, --end TEMPLATE        listing end template
198 -t, --tangle               strip doc-comments; emit runnable source
199 -o, --output FILE         write output here instead of stdout
200 -h, --help                 show this help
201
202 Templates substitute @LANG@ with the language and @CONT@ with
203 "firstnumber=last," on continuation blocks (empty on the first).
204 ]])
205 end
206
207 local function die(msg)
208   io.stderr:write("comment2tex: " .. msg .. "\n")
209   os.exit(1)
210 end
211
212 function M.main(argv)
213   local over = {}
214   local tangle = false
215   local input
216   local i = 1
217   local function value(flag)
218     i = i + 1
219     local v = argv[i]
220     if v == nil then die("missing value for " .. flag) end
221     return v
222   end
223   while i <= #argv do
224     local a = argv[i]
225     if a == "-h" or a == "--help" then
226       usage(io.stdout); return 0
227     elseif a == "-s" or a == "--style" then
228       over.style = value(a)
229     elseif a == "-w" or a == "--wrapper" then
230       over.wrapper = value(a)
231     elseif a == "-c" or a == "--comment" then
232       over.comment = value(a)
233     elseif a == "-l" or a == "--language" then
234       over.language = value(a)
235     elseif a == "-b" or a == "--begin" then
236       over.begin = value(a)
237     elseif a == "-e" or a == "--end" then
238       over.finish = value(a)
239     elseif a == "-t" or a == "--tangle" then
240       tangle = true

```

```

241     elseif a == "-o" or a == "--output" then
242         over.output = value(a)
243     elseif a == "--" then
244         input = argv[i + 1]; break
245     elseif a:sub(1, 1) == "-" and a ~= "--" then
246         die("unknown option: " .. a)
247     elseif input == nil then
248         input = a
249     else
250         die("unexpected argument: " .. a)
251     end
252     i = i + 1
253 end
254
255 if not input then
256     usage(io.stderr); return 1
257 end
258
259 local ok, err = pcall(function()
260     local o = M.resolve(M.new_opts(over))
261     local text = tangle and M.tangle_file(input, o)
262     or M.convert_file(input, o)
263     if over.output then
264         local fh, e = io.open(over.output, "w")
265         if not fh then
266             error("comment2tex: cannot open output: " .. tostring(e))
267         end
268         fh:write(text)
269         fh:close()
270     else
271         io.stdout:write(text)
272     end
273 end)
274 if not ok then die(tostring(err):gsub("^comment2tex: ", "")) end
275 return 0
276 end
277
278 if not LIBRARY_MODE then
279     os.exit(M.main(arg))
280 end
281
282 return M

```

6 The listings variant

Everything above — *The build* and *The converter, by example* — used the default verbatim renderer. When a document loads `listings`, `\ctxuselistings` switches the blocks that follow to it (and `\ctxuseverbatim` switches back). Below is one annotated Bash source both ways: first the verbatim default, then the same file through `listings`, which adds syntax highlighting. Each renderer numbers from one; only the presentation differs.

Verbatim (default): A short annotated Bash source. Double-hash lines become prose, everything else is code.

```
1 echo "building"
2 for f in *.tex; do
3   lualatex --interaction=nonstopmode "$f"
4 done
```

Listings (`\ctxuselistings`): A short annotated Bash source. Double-hash lines become prose, everything else is code.

```
1 echo "building"
2 for f in *.tex; do
3   lualatex --interaction=nonstopmode "$f"
4 done
```

6.1 Styling

The two modes differ in capability, not just appearance. The verbatim default is self-contained: numbered (a `\reset@font\scriptsize` gutter auto-sized to the file's line count), code in `\small\ttfamily`, tabs expanded — but no syntax highlighting and, like `ltxdoc`'s `macrocode`, no long-line breaking. `listings` mode gives that self-containment up for the full `listings` feature set: per-language highlighting, line breaking, and the whole `\lstset` vocabulary.

The converter bakes only the essentials into each `lstlisting` block — `language=<lang>`, `numbers=left`, and (on every block after the first) `firstnumber=last`, so a file's line numbers run continuously. Everything else is left to a document-wide `\lstset`. So that `listings` mode looks sensible with no configuration, `comment2tex` applies exactly one default, `\AtBeginDocument` and only when `listings` is actually loaded:

- `basicstyle` is `\ttfamily\small`, the monospace body font;
- `columns=fullflexible`, natural glyph widths rather than a rigid grid;
- `breaklines=true`, wrapping long lines (the verbatim renderer cannot);
- `showstringspaces=false`, no visible-space markers inside strings.

It sets nothing else — no colours, no keyword or comment styles — so the default is deliberately plain, close to the verbatim look.

To change it, add your own `\lstset` *after* `\begin{<document>}` (or in a begin-document hook registered after `comment2tex`); the four keys above are applied at begin-document, so a preamble `\lstset` of the same keys is overridden. Keys `comment2tex` never sets — keyword, comment and number styles, colours, and the like — may go anywhere and survive. `listings` has no `yaml` language; to highlight YAML, define one before the first `\includeyaml` (*Usage*); otherwise an empty one is installed and YAML falls back to the plain default.

7 Implementation

7.1 The L^AT_EX wrapper

Announce the package. `listings` is an *optional* companion, not a hard dependency: if the document has loaded it, the include macros emit `lstlisting` blocks and

a neutral default `\lstset` is applied; otherwise they fall back to a built-in numbered verbatim (below). The `\lstset` is deferred to `\AtBeginDocument` so it runs regardless of the order in which the document loads listings, and only when it does.

```

1 \NeedsTeXFormat{LaTeX2e}
2 \ProvidesPackage{comment2tex}
3 [2026/07/22 v1.1 Include annotated source files as LaTeX listings]
4 \AtBeginDocument{%
5   \ifpackageloaded{listings}{%
6     \lstset{basicstyle=\ttfamily\small,columns=fullflexible,%
7       breaklines=true,showstringspaces=false}%
8   }{}}

```

The name of the converter, overridable before loading if it is not on the current directory or $\TeX$ tree.

```
9 \providecommand\ctxscript{comment2tex.lua}
```

Under Lua $\TeX$, load the converter once into the global `comment2tex`, locating it through `kpse`.

```

10 \ifdefined\directlua
11   \directlua{
12     if not comment2tex then
13       local f = kpse.find_file("comment2tex.lua","lua")
14       or kpse.find_file("comment2tex.lua") or "comment2tex.lua"
15       comment2tex = assert(loadfile(f))("comment2tex")
16     end
17   }
18 \fi

```

`\ctxlisting` The verbatim fallback, used when `listings` is absent. It reuses the converter’s `plain` wrapper (which brackets each code block with `\ctxlisting%` and `\endctxlisting`) and mirrors the plain $\TeX$ wrapper’s line-by-line reader — the one hook `listings` and `fancyvrb` rely on and the kernel `verbatim` environment lacks — so every line can be numbered. `\ctx@lineno` is a count reset once per included file (by `\ctx@include`) and advanced *globally* per line, so it survives the group boundary between consecutive blocks: that reproduces `listings`’ `firstnumber=last`. `\ctx@num` styles the number like `ltxdoc`’s `macrocode` — `\reset@font\scriptsize`, right-aligned with `\llap` into a `\leftskip` gutter. `\ctxlistingtotal` sizes that gutter: the converter emits it once before the first block with a run of zeros matching the file’s code-line count, so the width fits the largest number exactly (a three-digit default, set in `\ctx@include`, covers hand-made fragments). `\ctx@uncatcode` switches to the verbatim regime (`\dospecials` makes the specials *and* the space category 12, preserving indentation); `\ctx@gobble` discards the remainder of the `\ctxlisting` line (its trailing %) before `\ctx@grab` collects real lines, exactly as the plain wrapper does.

```

19 \newcount\ctx@lineno
20 \newdimen\ctx@numindent
21 \providecommand\ctx@numstyle{\reset@font\scriptsize}
22 \def\ctxlistingtotal#1{\settowidth\ctx@numindent{\ctx@numstyle #1\ }}
23 \def\ctx@num{%
24   \global\advance\ctx@lineno\@ne
25   \llap{\ctx@numstyle\the\ctx@lineno\ }}
26 \def\ctx@uncatcode{\let\do\@makeother\dospecials}
27 \beginingroup

```

```

28 \catcode\^^M=\active%
29 \gdef\ctxlisting{%
30   \par\beginingroup
31   \ctx@uncatcode
32   \parindent=\z@ \leftskip=\ctx@numindent \small\ttfamily
33   \catcode\^^M=\active%
34   \ctx@gobble}%
35 \gdef\ctx@gobble#1^^M{\ctx@grab}%
36 \gdef\ctx@grab#1^^M{\ctx@checkend{#1}}%
37 \endgroup
38 \def\ctx@checkend#1{%
39   \def\ctx@tmp{#1}%
40   \ifx\ctx@tmp\ctx@endmark
41     \let\ctx@next\ctx@finish
42   \else
43     \leavevmode\ctx@num#1\par
44     \let\ctx@next\ctx@grab
45   \fi
46   \ctx@next}
47 \def\ctx@finish{\par\endgroup}
48 \beginingroup
49 \catcode\|=0 \catcode\|=12
50 \gdef\ctx@endmark{\endctxlisting}%
51 \endgroup

```

`\ctxuselistings` The rendering mode is switchable anywhere in the document and defaults to the
`\ctxuseverbatim` numbered-verbatim fallback, so a document needs neither `listings` nor any configuration to typeset a source. `\ctxuselistings` routes subsequent includes through `listings`; `\ctxuseverbatim` routes them back to the verbatim fallback. Both are ordinary declarations — honour $\TeX$ grouping, so a group or environment can switch mode locally. Requesting `listings` while the package is not loaded warns once and stays on verbatim.

```

52 \newif\ifctx@lst
53 \newif\ifctx@warned
54 \newcommand\ctxuselistings{\ctx@lsttrue}
55 \newcommand\ctxuseverbatim{\ctx@lstfalse}

```

`\ctx@include` Strip the extension, build the output name, pick the wrapper from the current mode — `lstlisting` in `listings` mode (falling back to the numbered-verbatim plain wrapper, with a one-time warning, when `listings` is not loaded), otherwise plain — generate it the best way the engine allows, then input it. The verbatim path resets the line counter once per file. `\ctx@stripdir` drops any directory part and `\ctx@strip` then keeps everything before the first dot, so a path such as `.github/workflows/build.yml` yields the base `build`; the sentinel dot still lets an extensionless name (a `Makefile`) keep its whole self.

```

56 \def\ctx@strip#1.#2\ctx@stop{#1}
57 \def\ctx@stripdir#1{\ctx@sd#1/\ctx@sdstop}
58 \def\ctx@sd#1/#2\ctx@sdstop{\ifx\relax#2\relax#1\else\ctx@sd#2\ctx@sdstop\fi}
59 \newcommand\ctx@include[2]{%
60   \edef\ctx@in{#2}%
61   \edef\ctx@base{\ctx@stripdir{#2}}%
62   \edef\ctx@base{\expandafter\ctx@strip\ctx@base.\ctx@stop}%
63   \edef\ctx@out{\ctx@base.c2t.tex}%

```

```

64 \ifctx@lst
65   \@ifpackageloaded{listings}%
66     {\def\ctx@wrap{lstlisting}}%
67     {\ifctx@warned\else \ctx@warnedtrue
68       \PackageWarning{comment2tex}{\string\ctxuselisting\space requested but
69         listings is not loaded; using the verbatim fallback}%
70       \fi
71       \def\ctx@wrap{plain}\global\ctx@lineno\z@\ctxlistingtotal{000}}%
72   \else
73     \def\ctx@wrap{plain}\global\ctx@lineno\z@\ctxlistingtotal{000}%
74   \fi
75   \ifdefined\directlua
76     \directlua{comment2tex.write("#1","\ctx@wrap",
77       "\luaescapestring{\ctx@in}","\luaescapestring{\ctx@out}")}%
78   \else
79     \ifdefined\pdfshellescape
80       \ifnum\pdfshellescape>0\relax
81         \immediate\write18{texlua \ctxscript\space
82           --style #1 --wrapper \ctx@wrap\space -o \ctx@out\space \ctx@in}%
83       \fi
84     \fi
85   \fi
86   \IfFileExists{\ctx@out}{\input{\ctx@out}}{%
87     \PackageError{comment2tex}{Converted file \ctx@out\space not found}%
88     {Use LuaLaTeX, run pdflatex with -shell-escape, or pre-build it:^^J%
89     texlua \ctxscript\space --style #1 --wrapper \ctx@wrap\space
90     -o \ctx@out\space \ctx@in}}%
91 }

```

`\includebash` The four public entry points fix the style. `listings` provides the `bash`, `Lua` and `make` `\includelua` languages itself, but ships no `yaml` one, so in `listings` mode `\includeyaml` first `\includeyaml` ensures a `yaml` language exists: if the document has loaded `listings` but not defined `\includemake` one, `\ctx@ensureyaml` installs an *empty* language so the emitted `language=yaml` resolves to the default listing style (no YAML-specific highlighting) instead of faulting. The test peeks at `listings`' own language store (`\lstlang@yaml$`); a document that defines its own `yaml` language is left untouched. In verbatim mode (the default) there is no language, so the guard is a no-op and never touches the `listings` internals.

```

92 \newcommand\ctx@ensureyaml{%
93   \ifctx@lst
94     \@ifpackageloaded{listings}{%
95       \@ifundefined{\@lst lang@yaml$}{\lstdefinelanguage{yaml}{}{}}{%
96       }{}%
97     \fi}
98 \newcommand\includebash[1]{\ctx@include{bash}{#1}}
99 \newcommand\includelua[1]{\ctx@include{lua}{#1}}
100 \newcommand\includeyaml[1]{\ctx@ensureyaml\ctx@include{yaml}{#1}}
101 \newcommand\includemake[1]{\ctx@include{make}{#1}}

```

7.2 The plain T_EX wrapper

Plain T_EX has neither `listings` nor the L^AT_EX file helpers, so the wrapper carries its own verbatim listing and uses primitive file tests. Control-sequence names are

letters only (no digits, no @).

Locate and load the converter under Lua_T_EX, exactly as the package does.

```

1 \def\ctxscript{comment2tex.lua}
2 \expandafter\ifx\csname directlua\endcsname\relax\else
3   \directlua{
4     if not comment2tex then
5       local f = kpse.find_file("comment2tex.lua","lua")
6       or kpse.find_file("comment2tex.lua") or "comment2tex.lua"
7       comment2tex = assert(loadfile(f))("comment2tex")
8     end
9   }
10 \fi

```

\ctxendmark The end-of-listing sentinel, stored as a string of category-12 characters so it can be compared against a verbatim line. A temporary escape character (I) lets us define it while the backslash is category 12 (“other”).

```

11 \begingroup
12 \catcode`\|=0 \catcode`\|=12
13 |gdef|ctxendmark{\endctxlisting}%
14 |endgroup

```

\ctxuncatcode Make the usual specials (and the space) category 12, the verbatim regime.

```

15 \def\ctxuncatcode{\def\do##1{\catcode`##1=12 }\dospecials}

```

\ctxlistingtotal The plain _T_EX listing carries no line numbers, so the converter’s gutter-sizing hint (emitted once before the first block) is a no-op here.

```

16 \def\ctxlistingtotal#1{}

```

\ctxlisting **\ctxlisting** (emitted by the converter as **\ctxlisting%**) opens a group, switches **\ctxgobble** to the verbatim regime with an active end-of-line, and starts grabbing lines.

\ctxgrab The trailing % on the begin line *cannot* be relied on to swallow that line end: **\ctxuncatcode** has already made % category 12 by the time the scanner reaches it, so it would be grabbed as a spurious first line. **\ctxgobble** therefore discards everything up to and including the first active $\sim$ (the remainder of the begin line) before **\ctxgrab** collects the first real line. **\ctxgrab** collects one line up to the active $\sim$ and hands it on.

```

17 \begingroup
18 \catcode`\~M=\active%
19 |gdef|ctxlisting{%
20   \par\begingroup%
21   \ctxuncatcode%
22   \parindent=0pt \tt%
23   \catcode`\~M=\active%
24   \ctxgobble}%
25 |gdef|ctxgobble#1~M{\ctxgrab}%
26 |gdef|ctxgrab#1~M{\ctxcheckend{#1}}%
27 \endgroup

```

\ctxcheckend Compare the grabbed line with the sentinel: if equal, finish; otherwise typeset it and loop. **\ctxnext** defers the recursion until after **\fi**.

```

28 \def\ctxcheckend#1{%
29   \def\ctxtmp{#1}%

```

```

30 \ifx\ctxtmp\ctxendmark
31 \let\ctxnext\ctxfinish
32 \else
33 \leavevmode#1\par
34 \let\ctxnext\ctxgrab
35 \fi
36 \ctxnext}
37 \def\ctxfinish{\par\endgroup}

\ctxinclude Mirror of the LATEX \ctx@include: derive the output name, generate it in process
(LuaTEX) or via shell escape (pdfTEX), then input it, erroring if it is still missing.
38 \def\ctxstrip#1.#2\ctxstop{#1}
39 \def\ctxstripdir#1{\ctxsd#1/\ctxsdstop}
40 \def\ctxsd#1/#2\ctxsdstop{\ifx\relax#2\relax#1\else\ctxsd#2\ctxsdstop\fi}
41 \def\ctxinclude#1#2{%
42 \edef\ctxin{#2}%
43 \edef\ctxbase{\ctxstripdir{#2}}%
44 \edef\ctxbase{\expandafter\ctxstrip\ctxbase.\ctxstop}%
45 \edef\ctxout{\ctxbase.c2t.tex}%
46 \expandafter\ifx\csname directlua\endcsname\relax
47 \expandafter\ifx\csname pdfshellescape\endcsname\relax\else
48 \ifnum\pdfshellescape>0
49 \immediate\write18{texlua \ctxscript\space
50 --style #1 --wrapper plain -o \ctxout\space \ctxin}%
51 \fi
52 \fi
53 \else
54 \directlua{comment2tex.write("#1","plain",
55 "\luaescapestring{\ctxin}","\luaescapestring{\ctxout}")}%
56 \fi
57 \openin0=\ctxout\relax
58 \ifeof0
59 \closein0
60 \errmessage{comment2tex: \ctxout\space not found
61 (use luatex, -shell-escape, or pre-run texlua)}%
62 \else
63 \closein0
64 \input \ctxout\relax
65 \fi}

\includebash The public entry points.
\includelua 66 \def\includebash{\ctxinclude{bash}}
\includeyaml 67 \def\includelua{\ctxinclude{lua}}
\includemake 68 \def\includeyaml{\ctxinclude{yaml}}
69 \def\includemake{\ctxinclude{make}}

```

8 Testing

comment2tex carries a cross-engine test suite, `comment2tex-test.sh` — and, true to the package, that script is itself an annotated Bash source. What follows is the script, typeset by `\includebash` in the default verbatim mode: its `##` lines are the prose you read, everything else is the code that runs.

The suite is a development artefact and is not part of the distribution, so the listing appears only when the script is present next to the `.dtx`.

```
1 #!/usr/bin/env bash
```

8.1 What the suite covers

`comment2tex-test.sh` drives the converter and both wrappers through every supported $\text{\TeX}$ route, so a change cannot quietly break one path while another keeps working. Each route is exercised end to end — annotated source to `.c2t.tex` fragment to *PDF*:

- `texlua` — the converter as a command-line program.
- $\text{\Lua\TeX}$ — in process through `\directlua` (no shell escape, no pre-run).
- $\text{\pdf\TeX}$ — shell escape, and a separate run over pre-built fragments.
- plain `luatex` and `pdftex` — the plain $\text{\TeX}$ wrapper.
- $\text{\Lua\TeX}$ on `comment2tex.dtx` itself — the self-documenting `\includelua` build of this manual.

A missing engine is skipped, never failed; any engine that runs and fails makes the whole suite exit non-zero. `TEXMFHOME` is redirected to an empty tree so a personal listings configuration cannot leak into the run.

```
2 set -uo pipefail
3
4 SRC="$(cd "$(dirname "$0")" && pwd)"
5 WORK="$(mktemp -d "${TMPDIR:-/tmp}/comment2tex-test.XXXXXX")"
6 export TEXMFHOME="$WORK/texmf-empty"
7 mkdir -p "$TEXMFHOME"
8
9 pass=0 fail=0 skip=0
10 failed_names=()
11
```

8.2 Harness helpers

`log` prints a line; `have` reports whether an engine is on the `PATH`.

```
12 log() { printf '%s\n' "$*"; }
13 have() { command -v "$1" >/dev/null 2>&1; }
14
```

`run <name> <output> -- <cmd>...` runs `<cmd>` with stdin closed and passes only when it exits zero *and* leaves a non-empty `<output>` *PDF*; on failure it prints the first $\text{\TeX}$ error.

```
15 run() {
16   local name="$1" out="$2"; shift 2
17   [ "$1" = "--" ] && shift
18   rm -f "$out"
19   if "$@" </dev/null >"$WORK/last.log" 2>&1 && [ -s "$out" ]; then
20     log "PASS $name"
21     pass=$((pass + 1))
22   else
23     log "FAIL $name (exit $?)"

```

```

24     sed -n '/^! /,+4p' "$WORK/last.log" | sed 's/^/      /' | head -20
25     fail=$((fail + 1))
26     failed_names+=("$name")
27 fi
28 }
29
30 skip() { log "SKIP $name -- $1"; skip=$((skip + 1)); }
31
32 C2T="texlua $WORK/comment2tex.lua"
33
34     Common engine flags, factored out so the driver runs stay within one line: $NS
35     is nonstop interaction, $OPTS adds halt-on-error (the self-documenting build omits
36     the halt so a warning cannot stop it).
37     NS="-interaction=nonstopmode"
38     OPTS="$NS -halt-on-error"
39
40

```

8.3 Fixtures

Everything runs in a scratch copy of the sources, and the wrappers are extracted from the .dtx first, so the suite tests exactly what docstrip ships rather than a stale build. A wrapper that will not build is a hard error, not a skip.

```

41 log "workspace: $WORK"
42 cp -r "$SRC/comment2tex.lua" "$SRC/comment2tex.dtx" "$SRC/comment2tex.ins" \
43     "$SRC/Makefile" "$SRC/comment2tex-test.sh" "$SRC/.github" "$WORK"/ || {
44     log "FATAL: cannot find comment2tex sources next to this script"; exit 2; }
45
46 cd "$WORK" || exit 2
47
48 if have tex; then
49     if tex comment2tex.ins </dev/null >"$WORK/ins.log" 2>&1 \
50         && [ -s comment2tex.sty ] && [ -s comment2tex.tex ]; then
51         log "PASS docstrip extraction (comment2tex.sty, comment2tex.tex)"
52         pass=$((pass + 1))
53     else
54         log "FAIL docstrip extraction"
55         tail -5 "$WORK/ins.log" | sed 's/^/      /'
56         fail=$((fail + 1)); failed_names+=("docstrip extraction")
57     fi
58 else
59     log "FATAL: 'tex' is required to extract the wrappers"; exit 2
60 fi
61
62

```

Three tiny annotated fixtures, one per style, kept trivial so they typeset under both `article` and plain `TeX`. Bash and YAML use the `##` doc-prefix, Lua uses `---`. The YAML fixture opens with a `---` document-start marker — a code line that exercises whatever `literate` rule a `yaml` language applies to it.

```

63 printf '%s\n' \
64     '## Demonstration Bash source.' \
65     '## Double-hash lines become prose.' \
66     'echo "hello"' \
67     'ls -la' \
68     '## A second paragraph, then more code.' \

```

```

64 'exit 0' > demo-bash.sh
65 printf '%s\n' \
66 '--- Demonstration Lua source.' \
67 '--- Triple-dash lines become prose.' \
68 'local x = 1' \
69 'print(x)' \
70 '--- Done.' > demo-lua.lua
71 printf '%s\n' \
72 '## Demonstration YAML source.' \
73 '## Double-hash lines become prose.' \
74 '---' \
75 'key: value' \
76 'list:' \
77 ' - a' \
78 ' - b' \
79 '## Done.' > demo-yaml.yml
80

```

8.4 Driver documents

One L^AT_EX (and one plain T_EX) driver per behaviour under test. The default driver loads no listings, so the include macros must fall back to the built-in numbered verbatim without error.

```

81 cat > doc-latex.tex <<'EOF'
82 \documentclass{article}
83 \usepackage{comment2tex}
84 \begin{document}
85 \includebash{demo-bash.sh}
86 \includelua{demo-lua.lua}
87 \includeyaml{demo-yaml.yml}
88 \end{document}
89 EOF
90

```

In listings mode with no yaml language defined, `\includeyaml` must install its own empty one and typeset under the default style.

```

91 cat > doc-latex-listings.tex <<'EOF'
92 \documentclass{article}
93 \usepackage{listings}
94 \usepackage{comment2tex}
95 \begin{document}
96 \ctxuselistings
97 \includebash{demo-bash.sh}
98 \includelua{demo-lua.lua}
99 \includeyaml{demo-yaml.yml}
100 \end{document}
101 EOF
102

```

In listings mode *with* a self-contained yaml language (no external package — `comment2tex` depends on no yaml highlighter), `\includeyaml` must leave that definition untouched and use it.

```

103 cat > doc-latex-yaml.tex <<'EOF'
104 \documentclass{article}
105 \usepackage{listings}

```

```

106 \usepackage{comment2tex}
107 \lstdefinelanguage{yaml}{
108   comment=[1]{\#},
109   morestring=[b]',
110   morestring=[b]",
111   literate={---}{\bfseries-{}-{}-}3
112 }
113 \begin{document}
114 \ctxuselistings
115 \includeyaml{demo-yaml.yaml}
116 \end{document}
117 EOF
118

```

The switch driver renders one file verbatim, then listings, then verbatim again, exercising `\ctxuselistings` and `\ctxuseverbatim` in a single run.

```

119 cat > doc-latex-switch.tex <<'EOF'
120 \documentclass{article}
121 \usepackage{listings}
122 \usepackage{comment2tex}
123 \begin{document}
124 Default verbatim:\par
125 \includebash{demo-bash.sh}
126 \ctxuselistings
127 Now listings:\par
128 \includebash{demo-bash.sh}
129 \ctxuseverbatim
130 Back to verbatim:\par
131 \includeyaml{demo-yaml.yaml}
132 \end{document}
133 EOF
134

```

The plain $\TeX$ driver `\inputs comment2tex.tex` directly.

```

135 cat > doc-plain.tex <<'EOF'
136 \input comment2tex.tex
137 \includebash{demo-bash.sh}
138 \includelua{demo-lua.lua}
139 \includeyaml{demo-yaml.yaml}
140 \bye
141 EOF
142

```

`gen_frgs` (*wrapper*) pre-builds the three fragments for the separate-run routes; `clean_frgs` deletes them so an in-process route cannot silently reuse a stale fragment.

```

143 clean_frgs() { rm -f demo-bash.c2t.tex demo-lua.c2t.tex demo-yaml.c2t.tex; }
144 gen_frgs() { # gen_frgs WRAPPER
145   $C2T --style bash --wrapper "$1" -o demo-bash.c2t.tex demo-bash.sh &&
146   $C2T --style lua --wrapper "$1" -o demo-lua.c2t.tex demo-lua.lua &&
147   $C2T --style yaml --wrapper "$1" -o demo-yaml.c2t.tex demo-yaml.yaml
148 }
149

```

8.5 The converter as a command-line program

The CLI must emit an `lstlisting` block per style, the plain wrapper's `\ctxlisting` and `\endctxlisting` brackets, and — with `--tangle` — byte-for-byte what the `sed` it replaces would, for every style.

```
150 name="texlua CLI (lstlisting + plain output)"
151 if have texlua; then
152   if $C2T --style bash demo-bash.sh \
153       | grep -q '\\begin{lstlisting}\[language=bash' &&
154     $C2T --style lua demo-lua.lua \
155       | grep -q '\\begin{lstlisting}\[language={\[5.3\]Lua}' &&
156     $C2T --style yaml demo-yaml.yaml \
157       | grep -q '\\begin{lstlisting}\[language=yaml' &&
158     $C2T --style make Makefile \
159       | grep -q '\\begin{lstlisting}\[language=make' &&
160     $C2T --style yaml --wrapper plain demo-yaml.yaml \
161       | grep -q '\\ctxlisting' &&
162     $C2T --style yaml --wrapper plain demo-yaml.yaml \
163       | grep -q '\\endctxlisting'; then
164       log "PASS $name"; pass=$((pass + 1))
165     else
166       log "FAIL $name"; fail=$((fail + 1)); failed_names+=("$name")
167     fi
168
169   # --tangle strips doc-comments, style-aware: same output as the old sed.
170   name="texlua CLI --tangle (== sed, per style)"
171   if diff -q <(sed '/^##/d' demo-bash.sh) \
172       <($C2T --style bash --tangle demo-bash.sh) >/dev/null &&
173     diff -q <(sed '/^---/d' demo-lua.lua) \
174       <($C2T --style lua --tangle demo-lua.lua) >/dev/null &&
175     diff -q <(sed '/^##/d' demo-yaml.yaml) \
176       <($C2T --style yaml --tangle demo-yaml.yaml) >/dev/null; then
177       log "PASS $name"; pass=$((pass + 1))
178     else
179       log "FAIL $name"; fail=$((fail + 1)); failed_names+=("$name")
180     fi
181
182   # Parity with the original shell scripts, when they are present.
183   if [ -x "$SRC/bash2tex.sh" ]; then
184     name="texlua CLI parity vs bash2tex.sh"
185     if diff -q <("$SRC/bash2tex.sh" "$SRC/bash2tex.sh") \
186         <($C2T --style bash "$SRC/bash2tex.sh") >/dev/null; then
187       log "PASS $name"; pass=$((pass + 1))
188     else
189       log "FAIL $name"; fail=$((fail + 1)); failed_names+=("$name")
190     fi
191   fi
192 else
193   skip "texlua not installed"
194 fi
195
```

8.6 The L^AT_EX wrapper

The same three fixtures are typeset through every L^AT_EX route — verbatim default, both listings paths, the mid-document switch, and pdfL^AT_EX by shell escape and by separate run — each passing only on a non-empty *PDF*.

```
196 name="LuaLaTeX in-process (verbatim default, no listings)"
197 if have lualatex; then
198   clean_fragments
199   run "$name" doc-latex.pdf -- lualatex $OPTS doc-latex.tex
200 else skip "lualatex not installed"; fi
201
202 name="LuaLaTeX in-process (listings mode, yaml fallback language)"
203 if have lualatex; then
204   clean_fragments
205   run "$name" doc-latex-listings.pdf -- lualatex $OPTS doc-latex-listings.tex
206 else skip "lualatex not installed"; fi
207
208 name="LuaLaTeX in-process (listings mode, document-defined yaml language)"
209 if have lualatex; then
210   rm -f demo-yaml.c2t.tex
211   run "$name" doc-latex-yaml.pdf -- lualatex $OPTS doc-latex-yaml.tex
212 else skip "lualatex not installed"; fi
213
214 name="LuaLaTeX in-process (mid-document verbatim<->listings switch)"
215 if have lualatex; then
216   clean_fragments
217   run "$name" doc-latex-switch.pdf -- lualatex $OPTS doc-latex-switch.tex
218 else skip "lualatex not installed"; fi
219
220 name="pdfLaTeX -shell-escape (verbatim default)"
221 if have pdflatex; then
222   clean_fragments
223   run "$name" doc-latex.pdf -- pdflatex -shell-escape $OPTS doc-latex.tex
224 else skip "pdflatex not installed"; fi
225
226 name="pdfLaTeX separate-run, verbatim (no shell escape)"
227 if have pdflatex && have texlua; then
228   clean_fragments
229   gen_fragments plain
230   run "$name" doc-latex.pdf -- pdflatex $OPTS doc-latex.tex
231 else skip "pdflatex or texlua not installed"; fi
232
233 name="pdfLaTeX separate-run, listings (no shell escape)"
234 if have pdflatex && have texlua; then
235   clean_fragments
236   gen_fragments lstlisting
237   run "$name" doc-latex-listings.pdf -- pdflatex $OPTS doc-latex-listings.tex
238 else skip "pdflatex or texlua not installed"; fi
239
```

8.7 The plain T_EX wrapper

The plain wrapper has no listings, so it is driven under `luatex` (in process) and `pdftex` (shell escape and separate run).

```
240 name="plain luatex in-process"
241 if have luatex; then
242   clean_fragments
243   run "$name" doc-plain.pdf -- luatex $OPTS doc-plain.tex
244 else skip "luatex not installed"; fi
245
246 name="plain pdftex -shell-escape"
247 if have pdftex; then
248   clean_fragments
249   run "$name" doc-plain.pdf -- pdftex -shell-escape $OPTS doc-plain.tex
250 else skip "pdftex not installed"; fi
251
252 name="plain pdftex separate-run (no shell escape)"
253 if have pdftex && have texlua; then
254   clean_fragments
255   gen_fragments plain
256   run "$name" doc-plain.pdf -- pdftex $OPTS doc-plain.tex
257 else skip "pdftex or texlua not installed"; fi
258
```

8.8 The documentation itself

Finally the suite builds `comment2tex.dtx` — the very manual you are reading — which `\includeluas` the converter and `\includebashes` this script, so a break in either self-documenting include fails the build.

```
259 name="LuaLaTeX builds comment2tex.dtx"
260 if have lualatex; then
261   rm -f comment2tex.c2t.tex
262   run "$name" comment2tex.pdf -- lualatex $NS comment2tex.dtx
263 else skip "lualatex not installed"; fi
264
```

8.9 Summary

The tally is printed; a non-zero `fail` keeps the workspace for inspection and exits non-zero, otherwise the scratch tree is removed.

```
265 log ""
266 log "-----"
267 log "PASS: $pass  FAIL: $fail  SKIP: $skip"
268 if [ "$fail" -gt 0 ]; then
269   log "Failures: ${failed_names[*]}"
270   log "Workspace kept for inspection: $WORK"
271   exit 1
272 fi
273 rm -rf "$WORK"
274 log "All engine tests passed."
```

9 CI/CD

Two GitHub Actions workflows under `.github/workflows/` keep the package honest, and — like the `Makefile` and the test suite — they are annotated YAML, typeset here by `\includeyaml` (the fourth include macro, shown at last, on the package’s own continuous-integration setup). *build* tests and typesets on every push and pull request; *publish* bundles for CTAN and drafts a release on every tag. Both are development artefacts, not shipped, so each listing appears only when its file is present.

9.1 The build workflow

`build.yml` runs on every push to the default branch and on every opened or synchronised pull request. It runs the cross-engine test suite and typesets the manual inside the Xerdi $\text{\TeX}$ Live container, then archives the PDF so a failed run can still be inspected. Unlike the other XDP packages it fetches no shared `texmf` tree — a stock $\text{\TeX}$ Live is enough.

```
1 name: build
2
3 on:
4   push:
5     branches: [master, main]
6   pull_request:
7     types: [opened, synchronize]
8
```

One job on `ubuntu-latest`: check out the sources, run `make test doc` in the container, and upload the manual — `if: always()`, so the PDF survives even when a step failed.

```
9 jobs:
10   build:
11     runs-on: ubuntu-latest
12     steps:
13       - name: Check out the repository
14         uses: actions/checkout@v4
15         with:
16           path: comment2tex
17           fetch-tags: true
18           fetch-depth: 0
19       - name: Test and build in the TeX Live container
20         uses: addnab/docker-run-action@v3
21         with:
22           image: maclotsen/texlive:with-gf
23           shell: bash
24           options: --rm -i -v ${GITHUB_WORKSPACE}:/build
25           run: |
26             git config --global --add safe.directory /build/comment2tex
27             make -C comment2tex test doc
28       - name: Archive the manual
29         uses: actions/upload-artifact@v4
30         if: ${{ always() }}
31         with:
32           name: comment2tex
33           path: ${GITHUB_WORKSPACE}/comment2tex/comment2tex.pdf
```

9.2 The publish workflow

`publish.yml` runs when a tag is pushed. It rebuilds the CTAN bundle in the container and opens a *draft* GitHub release with the zip and the PDF attached, for the maintainer to review, correct and publish — or discard. A release is never made public automatically.

```
1 name: publish
2
3 on:
4   push:
5     tags:
6       - '*'
7
8 jobs:
9   publish:
10    runs-on: ubuntu-latest
11    steps:
12      - name: Check out the repository
13        uses: actions/checkout@v4
14        with:
15          path: comment2tex
16          fetch-tags: true
17          fetch-depth: 0
18      - name: Build the CTAN bundle
19        uses: addnab/docker-run-action@v3
20        with:
21          image: maclotsen/texlive:with-gf
22          shell: bash
23          options: --rm -i -v ${github.workspace}}:/build
24          run: |
25            git config --global --add safe.directory /build/comment2tex
26            make -C comment2tex package
27      - name: Create the draft release
28        id: create_release
29        uses: actions/create-release@v1
30        env:
31          GITHUB_TOKEN: ${secrets.GITHUB_TOKEN}}
32        with:
33          tag_name: ${github.ref}}
34          release_name: Release ${github.ref_name}}
35          draft: true
36          body: |
37            Release for version ${github.ref_name}}
38      - name: 'Upload release asset: CTAN bundle'
39        uses: actions/upload-release-asset@v1
40        env:
41          GITHUB_TOKEN: ${secrets.GITHUB_TOKEN}}
42        with:
43          upload_url: ${steps.create_release.outputs.upload_url}}
44          asset_path: ${github.workspace}}/comment2tex/comment2tex.zip
45          asset_name: comment2tex-${github.ref_name}}.zip
46          asset_content_type: application/zip
47      - name: 'Upload release asset: manual'
48        uses: actions/upload-release-asset@v1
```

```
49     env:
50       GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
51     with:
52       upload_url: ${ steps.create_release.outputs.upload_url }
53       asset_path: ${ github.workspace }/comment2tex/comment2tex.pdf
54       asset_name: comment2tex-${ github.ref_name }.pdf
55       asset_content_type: application/pdf
```